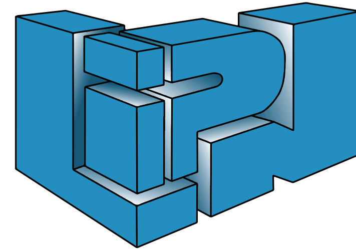
UNIVERSITÉ PARIS 13



THÈSE

présentée en vue de l'obtention du grade de

Docteur de l'Université Paris 13

Discipline: Informatique

Efficient Parametric Verification of Parametric Timed Automata

NGUYỄN Hoàng Gia

Soutenue publiquement le 28 août 2018

Jury:

Étienne ANDRÉ	MCF, HDR Université Paris 13, France	Co-directeur
Christophe FOUQUERÉ	Professeur Université Paris 13, France	Président
Didier LIME	MCF, HDR École Centrale de Nantes, France	Rapporteur
Wojciech PENCZEK	Professeur Polish Academy of Sciences, Poland	Rapporteur
Laure PETRUCCI	Professeur Université Paris 13, France	Co-directrice
Jun SUN	MCF Singapore University of Technology and Design, Singapore	Examinateur

Abstract

Critical real-time systems are becoming ubiquitous and are playing a vital role in our world. To provide guarantees that the system is behaving correctly, the correctness of these systems need to be verified before running. Besides functional checking, the timed behavior checking is also crucial. Indeed, the correctness of the systems also depends on the timing values or delays of internal operations that can be affected by the environment.

Verification techniques assure that software or hardware systems fully satisfy all their expected requirements. Most formal verification methods for timed systems guarantee the correctness of a timed system for the predefined timing values in its blueprint, but not for other undefined timing values which might occur by the environment change and lead to undesired system behaviors. Unfortunately, verifying such system for various timing values can be an obstacle and time-consuming. Therefore, by abstracting these specific timing values with parameters, many timing values of a system can be easily synthesized and checked at the same time: this technique is also known as parameter synthesis.

As a huge challenge for the verification, parameter synthesis techniques also suffer from the "state space explosion" problem, which is the explosion of the number of possible states while verifying a system formally.

First of all, we are interested in taking advantage of the capabilities of current distributed architectures, and parameter synthesis algorithms should be redefined and adapted to the distributed case. We propose in the thesis several distribution schemes that can accelerate our parameter synthesis procedures.

We also focus on studying the techniques such as symbolic verification, zone subsumption, etc. and how they affect the state space explosion problem. Then we introduce several smart state exploration techniques with some heuristics, in order to reduce the state space explosion. These techniques and heuristics are integrated into our new synthesis algorithms, and one of these algorithms is also extended in a distributed manner which gives an impressive performance in our benchmarks.

Furthermore, to achieve a reliable result we present an approach for detecting timed systems doing an infinite amount of actions in a finite time, which is known as the Zeno phenomenon in theory. In reality, it is infeasible and such counterexamples should always be avoided. Additionally, to detect the non-Zeno phenomenon on a large scale network model, we also distribute our approach on clusters. In the end, we introduce an algorithm to detect non-Zeno runs and its distributed version of it for large-scale models. At the time of writing this thesis, this is also the first work on non-Zeno parameter synthesis.

Keywords: Model checking, verification, real-time systems, parametric timed automata, parameter synthesis, distributed verification, parametric verification, exploration order, Zeno behaviors, distributed algorithms, IMITATOR.

Résumé

Les systèmes temps-réel critiques deviennent de plus en plus ubiquitaires et jouent un rôle majeur de nos jours. Pour garantir le bon comportement d'un système, leur correction doit être vérifiée avant les mises en service opérationnel. Outre la vérification fonctionnelle, la vérification du comportement temporel est également cruciale.

Les techniques de vérification garantissent que les systèmes logiciels ou matériels satisfont les contraintes attendues. La plupart des méthodes de vérification formelle pour les systèmes temporisés garantissent leur correction pour des valeurs prédéfinies des contraintes temporelles, mais pas pour d'autres valeurs non définies a priori, dues par exemple à un changement de l'environnement, et qui peuvent conduire à des comportements du système non désirés. Malheureusement, la vérification de tels systèmes pour différentes valeurs temporelles peut être un obstacle et coûteuse en temps. Ainsi, en s'abstrayant de valeurs temporelles spécifiques à l'aide de paramètres, de nombreuses valeurs temporelles d'un système peuvent être synthétisées et vérifiées simultanément: cette technique est la synthèse de paramètres.

Les techniques de synthèse de paramètres constituent un défi majeur pour la vérification. Elles souffrent du problème de l'explosion combinatoire de l'espace d'états, c'est-à-dire de la génération d'un nombre d'états considérable lors de la vérification formelle du système.

Dans un premier temps, nous nous sommes intéressés à tirer parti des spécificités des architectures informatiques distribuées modernes. Ainsi, les algorithmes de synthèse de paramètres ont dû être redéfinis et adaptés au cas distribué. Nous proposons dans cette thèse différents schémas de distribution pour accélérer les procédures de synthèse de paramètres.

Nous nous intéressons également à l'étude de techniques telles que la vérification symbolique, la subsomption, etc. et à leur impact sur l'explosion de l'espace d'états. Nous introduisons donc ensuite plusieurs heuristiques d'exploration afin de réduire l'explosion de l'espace d'états. Celles-ci sont intégrées à nos nouveaux algorithmes de synthèse de paramètres. L'un de ces algorithmes est étendu de manière distribuée, conduisant à des performances spectaculaires dans nos expérimentations.

Enfin, pour obtenir un résultat réalisable, nous présentons une approche qui détecte de systèmes temporisés effectuant un nombre infini d'actions en un temps fini, connu sous le nom de phénomène Zeno. En pratique, de tels comportements ne peuvent pas avoir lieu, et ne constituent pas des contre-exemples effectifs. De plus, nous proposons une approche distribuée pour détecter des phénomènes non Zeno à large échelle. Nous introduisons un algorithme détectant des comportements non Zeno, ainsi que sa version distribuée. Au moment de l'écriture de cette thèse, ceci constitue les premiers résultats sur la synthèse de paramètres non Zeno.

Mots-clefs : Model checking, vérification, systèmes temps-réel, automates temporisés paramétré, synthèse de paramètres, vérification distribuée, vérification paramétrée, stratégie d'exploration, comportements Zeno, algorithmes distribués, IMITATOR.

Remerciements

À l'issue de la rédaction de cette recherche, je suis convaincu que la thèse est loin d'être un travail solitaire. En effet, cette thèse n'aurait pu exister dans sa forme actuelle sans les personnes avec qui j'ai eu l'occasion de travailler ou de rencontrer au cours de ces trois années.

Mes remerciements vont en premier lieu à mes directeurs de thèse Prof. Assoc. Étienne André [1] et Prof. Laure Petrucci [2], en qui j'ai trouvé des professeurs talentueux, compréhensifs, humains, toujours disponibles. Leurs conseils, leurs encadrements, leurs implications, leurs corrections, leurs critiques ont beaucoup apporté à mon travail. Tout ce travail n'aurait pas abouti sans leurs encouragements et leurs soutiens constants.

Je tiens à remercier Christophe Fouqueré [3] d'avoir accepté d'être président du jury. Un très grand merci à Didier Lime [4] et Wojciech Penczek [5] pour m'avoir fait l'honneur d'être les rapporteurs de cette thèse. Prof. Assoc. Sun Jun [6] qui aura été pour cette thèse bien plus qu'un examinateur.

Je voudrais m'exprimer ma gratitude envers Prof. Assoc. Camille Coti [7], Prof. Assoc. Sami Evangelista [8] et Prof. Jaco van de Pol [9] pour leurs renseignements pendant des travaux en communs.

Je m'adresse un merci affectueux à Prof. Assoc. Quan Thanh Tho [10] et son groupe SAVE qui m'ont, d'une manière ou d'une autre, encouragé à entreprendre une thèse. Je souhaiterais aussi m'adresser ma gratitude aux professeurs d'université Bordeaux et d'université Paris 6 qui m'ont donné les cours informatiques au Vietnam.

Enfin, je voudrais remercier ma famille qui a, en tout temps, fait preuve d'affection et de compréhension. C'est elle qui m'a offert les meilleures conditions de recherche. Ces remerciements seraient incomplets si je n'adressais pas à l'ensemble des membres de LIPN et à mes amis Vietnamiens pour leur soutien moral ainsi que pour la très bonne ambiance que j'ai toujours trouvée au centre.

[1] http://lipn.univ-paris13.fr/~andre/
[2] http://lipn.univ-paris13.fr/~petrucci/
[3] https://lipn.univ-paris13.fr/~fouquere/
[4] http://www.irccyn.ec-nantes.fr/~lime/
[5] http://www.ipipan.waw.pl/~penczek/
[6] people.sutd.edu.sg/~sunjun/
[7] http://lipn.univ-paris13.fr/~coti/
[8] http://lipn.univ-paris13.fr/~evangelista/
[9] http://wwwhome.ewi.utwente.nl/~vdpol/
[10] http://www.cse.hcmut.edu.vn/~qttho/

Contents

1 Introduction **9**
- 1.1 Context . . . 9
  - 1.1.1 Formal methods . . . 10
  - 1.1.2 Testing vs. formal verification . . . 10
  - 1.1.3 Model checking and verification . . . 11
  - 1.1.4 Real-time model checking . . . 12
  - 1.1.5 Parametric model checking . . . 12
- 1.2 Objectives . . . 13
- 1.3 Contributions . . . 14
- 1.4 Organization of the document . . . 16

2 Preliminary definitions **17**
- 2.1 Introduction . . . 18
- 2.2 Labeled transition systems . . . 19
- 2.3 Clocks, parameters and constraints . . . 19
  - 2.3.1 Clocks . . . 19
  - 2.3.2 Parameters . . . 19
  - 2.3.3 Constraints . . . 20
- 2.4 Timed Automata . . . 21
  - 2.4.1 Introduction . . . 21
  - 2.4.2 Syntax . . . 21
  - 2.4.3 Concrete semantics . . . 23
  - 2.4.4 Problems . . . 25
  - 2.4.5 Tools and applications . . . 26
- 2.5 Parametric timed automata . . . 27
  - 2.5.1 Introduction . . . 27
  - 2.5.2 Syntax . . . 27
  - 2.5.3 Concrete semantics . . . 29
  - 2.5.4 Symbolic semantics . . . 29
  - 2.5.5 Subsumption abstraction . . . 31
  - 2.5.6 Problems . . . 32
  - 2.5.7 Tools and applications . . . 32
  - 2.5.8 Related formalisms . . . 33

2.6 System property specification . . . 33
2.7 Temporal logic . . . 34
2.8 State of the art . . . 35
2.8.1 Decision and computation problems . . . 35
2.8.2 Decidability of PTA . . . 37
2.9 Parameter synthesis . . . 39
2.9.1 The good parameters problem . . . 39
2.9.2 The Inverse problem . . . 40
2.9.3 The Inverse Method . . . 40
2.9.4 The Behavioral Cartography . . . 41
2.9.5 EF-problems . . . 42
2.10 IMITATOR . . . 43
2.11 Efficient verification . . . 43
2.11.1 On-the-fly verification . . . 43
2.11.2 Abstracted verification . . . 44
2.11.3 Compositional verification . . . 44
2.12 Parallel computing . . . 44

3 Distributed verification of parametric real-time systems 53
3.1 Introduction . . . 53
3.2 Static domain decomposition . . . 55
3.3 Master-worker point distribution algorithms . . . 57
3.3.1 Principle: master-worker . . . 57
3.3.2 An abstract algorithm for the master . . . 58
3.3.3 Sequential point distribution . . . 59
3.3.4 Random + sequential point distribution . . . 60
3.3.5 Shuffle point distribution . . . 61
3.4 Dynamic domain decomposition . . . 62
3.4.1 Master algorithm . . . 63
3.4.2 Worker algorithm . . . 64
3.4.3 An additional heuristic . . . 65
3.5 Experiments . . . 66
3.6 Conclusion . . . 72

4 Reachability preservation based parameter synthesis 73
4.1 Introduction . . . 73
4.2 Solving the EF-emptiness problem using reachability preservation 74
4.2.1 Undecidability of the preservation of reachability . . . 74
4.2.2 Parameter synthesis preserving the reachability . . . 76
4.2.3 EF-synthesis using PRP . . . 79
4.3 Towards distributed parameter synthesis . . . 80
4.4 Experimental comparison . . . 81
4.5 Conclusion . . . 84

5 Efficient synthesis using optimized state exploration strategies 85
5.1 Introduction 85
5.2 Parametric zone inclusion algorithm 87
5.3 Parametric ranking strategy 89
5.4 Parametric priority strategy 91
5.5 Experimental evaluation 94
5.5.1 Symbolic state merging 95
5.5.2 Comparison 95
5.5.3 Final interpretation 96
5.6 Conclusion 99

6 Layered and Collecting NDFS with Subsumption for Parametric Timed Automata 104
6.1 Introduction 104
6.2 Preserving accepting runs with subsumption 106
6.3 Parametric timed nested depth-first search with subsumption . . 107
6.3.1 NDFS with subsumption for PTA 107
6.3.2 Early pruning of the red search 110
6.3.3 Starting the red search early: A layered NDFS 110
6.4 Collecting NDFS for parameter synthesis 112
6.5 Experiments 112
6.5.1 Implementation 114
6.5.2 Experimental results 114
6.6 Conclusion 117

7 Parametric model checking under non-Zenoness assumption 118
7.1 Introduction 119
7.2 Undecidability of the non-Zeno emptiness problem 120
7.3 CUB-parametric timed automata 121
7.3.1 CUB timed automata 121
7.3.2 Parametric clock upper bounds 122
7.3.3 CUB parametric timed automata 123
7.3.4 CUB PTA detection 124
7.3.5 Transforming a PTA into a disjunctive CUB-PTA 127
7.4 Zeno-free cycle synthesis in CUB-PTAs 129
7.5 Distributing non-Zeno parametric model checking 137
7.5.1 Master algorithms 137
7.5.2 Worker algorithm 138
7.5.3 Handling the case of a network of PTAs 140
7.6 Experiments 141
7.6.1 Evaluation of the non-distributed version 141
7.6.2 Evaluation of the distributed version 144
7.7 Conclusion 145

8 **Conclusion and perspectives** **147**
8.1 Summary of the thesis . . . 147
8.2 Perspectives . . . 149

A **Appendix** **165**
A.1 Decidability . . . 165
A.1.1 Turing machine . . . 165
A.1.2 Halting problem . . . 166
A.1.3 Decidable and undecidable problems . . . 167
A.1.4 Reducibility . . . 167
A.2 Two-counter machine . . . 168

B **Distributed verification of parametric real-time systems** **169**
B.1 Existing algorithms . . . 169
B.1.1 The Inverse Method algorithm . . . 169
B.1.2 The Behavioral Cartography algorithm . . . 171
B.2 Master-worker point distribution algorithms . . . 171
B.2.1 Sequential point distribution: initialization algorithm . . . 171
B.2.2 Random point distribution: initialization algorithm . . . 171
B.2.3 Shuffle point distribution: algorithms . . . 172

C **Reachability preservation based parameter synthesis for timed automata** **173**
C.1 Reachability synthesis algorithm . . . 173
C.2 Proof of Proposition 4.2.3 . . . 174

List of Figures

1.1 The concept of model checking 11

2.1 Example of labeled transition system 19
2.2 Example of timed automaton 22
2.3 Example of network of parametric timed automata 24
2.4 Example of concrete run for the TA $\mathcal{A}$ in Fig. 2.3c 25
2.5 Example of trace for the TA $\mathcal{A}$ in Fig. 2.3c 25
2.6 Example of trace set for the TA $\mathcal{A}$ in Fig. 2.3c 26
2.7 Examples of parametric timed automaton and parametric timed Büchi automaton 47
2.8 Example of network of parametric timed automata 48
2.9 Example of symbolic run for the PTA $\mathcal{A}$ in Fig. 2.8c 48
2.10 Example of parametric zone graph PZG for the PTA $\mathcal{A}$ in Fig. 2.8c 49
2.11 Example of parametric zone graph PZG for the PBTA in Fig. 2.7b 49
2.12 Example of trace for the PTA $\mathcal{A}$ in Fig. 2.8c 49
2.13 Example of trace set for the PTA $\mathcal{A}$ in Fig. 2.8c 50
2.14 An example of parameter domain in multi-dimensions (hyperrectangle or polyhedron) 50
2.15 Graphical example 50
2.16 Examples of graphical behavioral cartographies in 2 dimensions . 51
2.17 EFsynth algorithm 51
2.18 An example of a PTA $\mathcal{A}_1$ [JLR15] 51
2.19 Functional view of IMITATOR 52

3.1 Graphical representations and challenges 56
3.2 Examples of graphical behavioral cartographies in 2 dimensions . 56
3.3 Sequential algorithm illustration 59
3.4 Random algorithm illustration 60
3.5 Shuffle algorithm illustration 61
3.6 Subdomain algorithm illustration 62
3.7 Subdomain algorithm with splitting process illustration 63
3.8 Experiments: execution time and speedup (1/3) 69
3.9 Experiments: execution time and speedup (2/3) 70
3.10 Experiments: execution time and speedup (3/3) 71

4.1 Undecidability of PREACH-emptiness: PTA $\mathcal{A}$ 75
4.2 EF-synthesis using PRPC and EFsynth for $\mathcal{A}_1$ 80
4.3 Cartography output by PRPC for Sched1 81
4.4 Cartography output by PRPC for Sched2 with $a = 50$ 82

5.1 Examples 100
5.2 Parametric zone graphs of Fig. 5.1a where the number n in a state label $\mathbf{s}_n$ reflects the exploration order 101
5.3 PZG with parametric ranking strategy 102
5.4 Comparing our two strategies 102
5.5 Blowup example 103
5.6 Inefficiency in largest zone first like algorithms 103

6.1 The symbolic state space $\mathsf{PZG}(\mathcal{B})$ of the model in Fig. 2.7b without information on contraints 106

7.1 Undecidability of the non-Zeno emptiness problem 121
7.2 Examples: detection of and transformation into CUB-PTAs . . . 125
7.3 Transformed version of Fig. 7.2c 126
7.4 Examples of PTAs to illustrate the CUB concept 127
7.5 Non-Zeno synthesis process flowchart 135
7.6 Parametric zone graph examples of the disjunctive CUB-PTAs . 136

List of Algorithms

1 Worker algorithm . . . 58
2 Abstract algorithm for the master . . . 59
3 Sequential.$choosePoint()$ . . . 60
4 Random.$choosePoint()$ . . . 61
5 Subdomain: Master . . . 64
6 Subdomain: Worker n . . . 65

7 PRP($\mathcal{A}, v$) . . . 77
8 PRPC($\mathcal{A}$,D) . . . 79

9 State exploration with parametric zone inclusion . . . 88
10 Ranking by parametric zone size . . . 90
11 $init_rank(l, C)$ . . . 91
12 $max_rank((l, C), r)$ . . . 91
13 Parametric priority strategy algorithm . . . 93

14 Classical NDFS . . . 108
15 NDFS with subsumption checks and red prune of *dfsBlue* (in light-blue) and early pruning (in yellow). . . . 109
16 Layered NDFS . . . 111
17 Layered collecting NDFS . . . 113

18 CUBdetect($\mathcal{A}$) . . . 124
19 CUBtrans($\mathcal{A}$): Transformation into a CUB-PTA . . . 130
20 CUB-PTA non-Zeno synthesis algorithm synthNZ($\mathcal{A}$) . . . 134
21 distSynthNZ($\mathcal{A}$): Master . . . 139
22 distSynthNZ($\mathcal{A}$): worker n . . . 140

23 Inverse Method IM($\mathcal{A}, v$) . . . 170
24 Behavioral Cartography BC($\mathcal{A}, D$) . . . 171
25 Sequential.$initialize()$ . . . 171
26 Random.$initialize()$ . . . 171
27 Shuffle.$initialize()$ . . . 172
28 Shuffle.$choosePoint()$ . . . 172

29 Reachability synthesis EFsynth($\mathcal{A}, s, G, S$) 174

CHAPTER 1

Introduction

Contents

1.1 Context 9
- 1.1.1 Formal methods 10
- 1.1.2 Testing vs. formal verification 10
- 1.1.3 Model checking and verification 11
- 1.1.4 Real-time model checking 12
- 1.1.5 Parametric model checking 12

1.2 Objectives 13

1.3 Contributions 14

1.4 Organization of the document 16

1.1 Context

The evolution of information technology industry makes computers becoming ever more present in our daily lives and computer software is becoming more and more complicated. Hence, there is more chance for failure. Failure is unacceptable for critical systems such as those used in the medical industry, biology, e-commerce, aeronautics, etc.

For instance, self-driving cars become more popular and real-time processing is an important issue, especially when self-driving cars are moving fast and need to react very quickly. Any delay in processing can lead to a fatal crash. Even when a car accident happens, any failure or delay of the airbag inflation can also cause serious injury or even human loss.

In 1991, the American army fired a Patriot Missile to intercept an Iraqi missile. Interception failed because of an inaccurate time calculation due to clock drift, and the Iraqi missile destroyed an American army barracks. Consequently, 28 soldiers died and 100 were injured. The reason is a floating point rounding error deriving from the system's internal clock. This incident shows that the robustness of timed critical systems is very important.

Other catastrophic failures come from other computer bugs in history, such as the Ariane 5 rocket that exploded 40 seconds after takeoff in 1996 and a bug in the Intel Pentium II chip in 1994. These bugs cost more than 400 million dollars. Between 1985 and 1987, the overdose of X-rays in the Therac-25 accident caused the death of at least 2 patients, and 4 patients were given radiation overdose.

Moreover, whether a system is a software or a hardware system, the later the bug is found, the more it costs and a question is how we could find the time interval of functional safety or fault tolerant of real-time systems. This is also the goal of this thesis, which is to find these time intervals efficiently.

1.1.1 Formal methods

Formal methods [HBH+99], are understood broadly as a particular kind of mathematically based techniques for the specification, verification and development. They are methods used to check rigorously the correctness of hardware and software. While traditional testing and simulation can prove the presence of bugs (errors) with finite test cases, and do not guarantee the absence of bugs. In contrast, formal methods based on mathematics, logic and reasoning, aim to prove the absence of bugs. In general, formal design can be seen as a three-step process:

- Formal Specification: design of a rigorously specified mathematical model to describe the system.
- Formal Verification: exhaustive search of the state space of the model to prove its correctness w.r.t. the expected properties.
- Implementation: convert the specification into code by development.

1.1.2 Testing vs. formal verification

Software and hardware testing has played an important role in finding bugs in systems for past decades. Software testing is a process based on predefining sets of given inputs and expected outputs for a software system and detecting the differences between its actual outputs with the expected outputs. For hardware systems, where fabrication processes are costly, in order to make sure a system works correctly it is usually tested by simulating a model of the real-world system. But the challenge is that with nowadays complex systems, it is impossible to identify or predict bad behaviors or guarantee that systems will perform as they should. However, testing cannot assure that a given system is error-free.

In other words, formal methods such as formal specification, formal verification, etc. have been proved to be successful as an effective and automatic solution that can guarantee a given system is error-free while reducing the number of resources and time spent. Instead of checking the system with its expected outputs, formal verification or formal methods are based on mathematical foundations to reason about the correctness of the system under consideration during the design phase. The system and its desirable behaviors are formalized and represented mathematically and precisely as a state machine and properties respectively. Then a computer program called *model checker* checks that the system property is satisfied by the model or exhibits an example that shows it is not satisfied. The next section will describe model checking and model checkers in detail.

1.1.3 Model checking and verification

Verification technologies such as model checking [BK08] or theorem proving are two well established formal verification techniques, and this thesis will be dedicated to model checking, which has been proved to be a successful technique and has many applications in various domains both in academy and industry. More specifically, it is widely used in hardware and software analysis for networking and telecommunications, medical, aeronautic, biology industries, etc.

Furthermore, in 2007, an ACM Turing award which is the most prestigious award in computer science was given to Edmund Melson Clarke, Ernest Allen Emerson and Joseph Sifakis for their excellence in developing model checking into a highly effective verification technology, and it could be seen as a recent evidence of success of the model checking field.

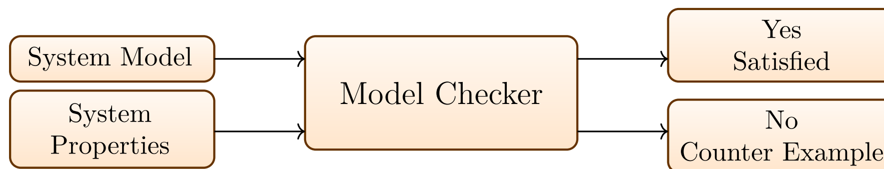


Figure 1.1 – The concept of model checking

In the model checking, the system to be verified is modeled as a transition system, and the property to be checked is expressed in a formal language such as e. g. temporal logics. Then, the transition system and the property are used as inputs to a model checker. The model checker explores exhaustively all reachable states of the transition system, to check whether the property is true, and the set of all explored states is called *state space*. Otherwise, if the property is not valid, the model checker exhibits a *counter-example* that violates the property. Based on this counter-example, we can detect the errors in the model and correct them. A diversity of properties can be verified by model-checking, such as deadlock freedom, invariants, request-response properties, etc. These properties can be

categorized into five classes: reachability, safety, liveness, deadlock-freeness, and fairness properties.

However, the main obstacle of the model checking approach is *state space explosion*: the transition graph usually becomes exponentially large based on the size of the system, which causes the exploration of the whole state space to become impracticable.

1.1.4 Real-time model checking

Real-time systems are notoriously difficult to design due to the complicated use of timing constraints, and must therefore be verified, e. g. using model checking. The correctness of real-time systems depends not only on its functional correctness but also on the timing values. As a consequence, the timing requirements in the system are therefore taken into consideration, and model checking is extended in order to verify real-time systems by adding timing information to the system model. Many formalisms were proposed to model and verify real-time systems e. g. timed Petri nets [Mer74] (an extension of Petri nets [Pet62]), timed communicating sequential processes (CSP) [Der00, OW02] (an extension of CSP [Hoa78]) and timed automata [AD90, AD94] (an extension of finite state automata). In Section 2.4 we will recall timed automata (TAs) which is one of the most popular formalisms used to verify real-time systems.

Unfortunately, due to frequent changes in the specification and the need of checking a wide range of timing values of the real-time systems, real-time model checking does not have the ability to adapt to these requirements. Technically, these limitations can be surpassed by parametric model checking which is introduced in Section 1.1.5.

1.1.5 Parametric model checking

Under time-critical conditions, the timing characteristics are vital in the design of real-time systems. In particular, when these systems are deployed in different environments, some unpredictable timing delays or unexpected behaviors might occur, which are not included in their design. Furthermore, from an industrial point of view, a new real-time system is often constructed or improved from its previous system design with some changes in the specification. Unfortunately, checking a specific system with given concrete timing values in real-time model checking is not sufficient. As a consequence, these requirements of frequently changing specifications are infeasible for real-time model checking (Section 1.1.4) in most cases, and this might be an explanation why formal methods are not as widespread as they could be.

To overcome the shortcomings of real-time model checking, parametric model checking was proposed to verify such real-time systems where some timing constants are unknown or uncertain. In a parametric model, we abstract the details of timing information of the real-time system by using parameters. It makes sense considering the delays as parameters, by this mean we can strengthen the system model and enrich the final result by finding parameter ranges for

which the system is correct or incorrect. Then the verification problems become parameter synthesis problems [AHV93]. Throughout this thesis, we will use parametric timed automata (PTA) [AHV93], an extension of TAs, for solving our synthesis problems. This formalism will be recalled in Section 2.5.

Additionally, many formalisms have been well adapted to the parametric case: parametric timed Petri nets [TLR09] (an extension of time Petri nets [Mer74]), parametric stateful timed CSP [ALSD14] (an extension of timed CSP [Der00, OW02]) and parametric timed automata (PTAs) [AHV93] (an extension of TAs [AD90, AD94]). In Section 2.5, we will present PTAs, which are used in this thesis and are also one of the most popular formalisms used for parametric model checking (other related formalisms are given in detail in Section 2.5.8). Section 2.9 will be dedicated to parameter synthesis problems.

Besides representing a range of timing values, other types of parameters can also be defined to represent the probabilities (uncertainty) [Geo14, KNSS99, Bea03], weights or cost (energy) [ADD+11, ADD+13], actions [KMP15, WR88], or discrete (processes) [Abd12, DSZ10, AD16].

1.2 Objectives

We have presented the context of real-time model checking for timed systems and benefits of using parametric model checking in Section 1.1.5. The general objective of the thesis aims at making the parametric verification procedures more efficient and reliable.

To this end, we follow mainly two approaches. In the first approach, we focus on developing efficient and reliable parametric checking algorithms, or semi-algorithms in case the termination of the algorithms is not guaranteed because of undecidable problems (Section 2.8.2). In the other, to make our parametric checking algorithms more efficient, we extend them in a distributed manner, so as to take advantage of high performance computing power available nowadays.

We indicate below the research avenues for the parameter synthesis approach and the objectives of this thesis are also summarized:

1. Although most problems of interest are undecidable for PTAs [And15], some (semi-)algorithms were proposed to tackle practical parameter synthesis (e. g. [ACEF09, KP12, JLR15, ABB+16]). Nevertheless, these algorithms still suffer from the state space explosion problem, where the number of states in the system increases exponentially. One of our objectives is to propose new efficient parameter synthesis algorithms. Then, by taking advantage of the capabilities of current distributed architectures, these algorithms will be redefined to be adapted to the distributed case.

2. Abstraction is proved as an efficient solution to the state space explosion problem [CC77, LGS+95]. One of the main abstractions widely used is the parametric zone graph, a symbolic semantics of a parametric timed automaton, which will be presented in Section 2.5. Besides that, we also

use parametric inclusion abstraction or zone subsumption techniques to reduce state space explosion.

Zone inclusion algorithm is an algorithm relied on zone subsumption technique to reduce state space explosion. Unfortunately, the zone inclusion algorithm is strongly affected by exploration order strategies and thus it might explore unnecessary states. Consequently, it will cause some issues such as time-consumption and high memory usage. In order to synthesize parameter valuations more efficiently, choosing an optimal search order for our state exploration algorithms is one of our goals.

In addition to synthesize parameter valuations of which a given state of a system is reachable, it is necessary to synthesize parameter valuations that a *good state* of a system is eventually reachable, this problem is also known as parameter synthesis for liveness properties. Then, by using zone subsumption technique, another goal is to synthesize parameter valuations for liveness properties efficiently.

3. The Zeno phenomenon in parametric model checking, also called Zeno behavior, is a non-realizable behavior where an infinite number of actions or discrete transitions can execute within a finite time in the model. Therefore, Zeno behavior is infeasible in reality and should be avoided. It is highly desirable when performing parametric model-checking that the parameter valuations satisfying the property should not allow the appearance of Zeno behaviors. Although there are many researches on the Zeno phenomenon for TAs ([BDR03, Tri99, TYB05, HS10, WSW+15]), there is none for PTAs.

1.3 Contributions

The main objectives of this thesis are to answer the research questions described in Section 1.2 and to develop adequate techniques or approaches. The contributions of this thesis are:

- In Chapter 3, we propose distributed parameter synthesis algorithms to compute Behavioral Cartography (BC) efficiently using parallel, distributed computing resources. Among the distributed algorithms we propose in this chapter, we show that the dynamic domain decomposition algorithm with heuristic Subdomain+H is faster than existing algorithms on large scale BC.

  This work has been achieved in collaboration with Étienne André and Camille Coti. It has been published in the 17th International Conference on Formal Engineering Methods - ICFEM'15. [ACN15]

- In Chapter 4, we introduce Parametric Reachability Preservation parameter synthesis algorithm (PRP) which is more efficient than BC. We then a compare the PRP with the previous BC. Furthermore, by reusing the

dynamic domain decomposition algorithm presented in Chapter 3, we propose a distributed version of PRP algorithm on behavioral cartography named PRPC which is faster and almost always outperforms the distributed BC algorithms.

This work has been done in collaboration with Étienne André, Giuseppe Lipari and Sun Youcheng. It has been published in the 7th NASA Formal Methods Symposium - NFM'15. [ALNS15]

- In Chapter 5, we propose some heuristic strategies based on breadth first search BFS for parametric zone inclusion algorithms called parametric priority strategy PRIOR and parametric ranking strategy RS, which mitigates state space explosion by reducing the *inefficient phenomenon* in state exploration.

  This work has been done in collaboration with Étienne André and Laure Petrucci. It has been published in the 22nd International Conference on Engineering of Complex Computer Systems - ICECCS'17. [ANP17]

- In Chapter 6, we introduces a series of variations of Nested Depth-First Search algorithm (NDFS) for PTAs. In fact, we introduce a new layered NDFS approach to Linear Temporal Logic (LTL) model checking and use this approach for our algorithms. In particular, we apply subsumption abstraction to PTA for the first time. This new layered approach and subsumption are added to our Layer NDFS and Layer Collecting NDFS algorithms called LayerNDFSsub and LayerCollectNDFSsub.

  This work has been done in collaboration with Laure Petrucci and Jaco van de Pol. It has been published in the 23nd International Conference on Engineering of Complex Computer Systems - ICECCS'18. [NPvdP18]

- In Chapter 7, we prove that the parameter synthesis problem for PTAs with non-Zenoness assumption is undecidable and we develop the semi-algorithms named CUBdetect and CUBtrans to solve the non-Zeno synthesis problem using CUB-PTA approach (clock upper bound parametric timed automata). In order to check large-scale models, we also design the distributed version of this CUBtrans semi-algorithm named distSynthNZ.

  - This work has been done in collaboration with Étienne André, Laure Petrucci and Sun Jun. It has been published in the 9th NASA Formal Methods Symposium - NFM'17. [ANPS17]
  - The work for the distributed version has been done in collaboration with Étienne André, Laure Petrucci and Sun Jun. This work will be submitted to a journal.

In order to evaluate the efficiency of these algorithms or semi-algorithms, we implemented all algorithms or semi-algorithms of each chapter in the IMITATOR tool [AFKS12], and we will also give an experimental validation together with a detailed contribution in each chapter.

1.4 Organization of the document

The structure of this document is organized in the following way. First of all, in Chapter 2, we give the formal definition of notions that will be frequently used later on.

After establishing some of the groundworks for the thesis, Chapter 3 is devoted to distributed parameter synthesis algorithms, where we introduce distributed behavioral cartography (BC) algorithms in detail. Then, Chapter 4 presents another parameter synthesis algorithm called parametric reachability preservation PRP which is more efficient than the previous BC algorithm. We also present the distributed algorithm of PRP which reuses the concepts of the distributed BC algorithm. In Chapter 5, we introduce state exploration order strategies for parametric zone inclusion, which contribute to alleviating the state space explosion problem. In Chapter 6, we propose some variations of Nested Depth-First Search algorithm (NDFS) making parameter synthesis for liveness properties more efficient. Then in Chapter 7, we introduce the Zeno phenomenon and approaches to detect non-Zeno runs in parametric timed automata. Our algorithm is then extended in distributed fashion and a performance evaluation is discussed.

Finally, in Chapter 8, concludes and presents directions for future research.

CHAPTER 2

Preliminary definitions

Contents

2.1 Introduction **18**
2.2 Labeled transition systems **19**
2.3 Clocks, parameters and constraints **19**
2.3.1 Clocks 19
2.3.2 Parameters 19
2.3.3 Constraints 20
2.4 Timed Automata **21**
2.4.1 Introduction 21
2.4.2 Syntax 21
2.4.3 Concrete semantics 23
2.4.4 Problems 25
2.4.5 Tools and applications 26
2.5 Parametric timed automata **27**
2.5.1 Introduction 27
2.5.2 Syntax 27
2.5.3 Concrete semantics 29
2.5.4 Symbolic semantics 29
2.5.5 Subsumption abstraction 31
2.5.6 Problems 32
2.5.7 Tools and applications 32
2.5.8 Related formalisms 33
2.6 System property specification **33**

2.7 Temporal logic **34**
2.8 State of the art **35**
2.8.1 Decision and computation problems 35
2.8.2 Decidability of PTA 37
2.9 Parameter synthesis **39**
2.9.1 The good parameters problem 39
2.9.2 The Inverse problem 40
2.9.3 The Inverse Method 40
2.9.4 The Behavioral Cartography 41
2.9.5 EF-problems 42
2.10 IMITATOR **43**
2.11 Efficient verification **43**
2.11.1 On-the-fly verification 43
2.11.2 Abstracted verification 44
2.11.3 Compositional verification 44
2.12 Parallel computing **44**

2.1 Introduction

This chapter introduces the basic knowledge of model checking and concepts related to this thesis. The remainder of the chapter is organized as follows:

1. In order to introduce the formalism and formal language are used in this thesis, from Section 2.2 to Section 2.5, show how labeled transition systems can be extended to parametric timed automata. Section 2.6 recalls some basic system properties and then Section 2.7 shows how these properties can be formalized in temporal logics.

2. The state of the art in Section 2.8 enumerates several decidable and undecidable problems of parametric model checking in the last few decades. Then the next Section 2.9 focuses only on problems and algorithms used repeatedly in this thesis.

3. Section 2.10 gives a brief introduction to IMITATOR, a parameter synthesis tool for real-time systems, in which our algorithms will be all implemented for evaluation purposes.

4. Section 2.11 recalls popular verification techniques that tackle state space explosion problem.

5. Section 2.12 covers basic knowledge of parallel computing and related concepts of it.

2.2 Labeled transition systems

We first introduce labeled transition systems, which will be used later in this chapter to represent the semantics of timed automata and parametric timed automata.

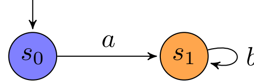


Figure 2.1 – Example of labeled transition system

Definition 2.2.1. A *labeled transition system* (LTS) is a quadruple $\mathcal{LTS} = (\Sigma, S, S_0, \Rightarrow)$, with Σ a set of symbols, S a set of *states*, $S_0 \subseteq S$ a set of *initial states*, and $\Rightarrow \in S \times \Sigma \times S$ a *transition relation*. We write $s \overset{a}{\Rightarrow} s'$ for $(s, a, s') \in \Rightarrow$. A *run* (of length m) of $\mathcal{LTS}$ is an alternating sequence of states $s_i \in S$ and symbols $a_i \in \Sigma$ of the form $s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} s_m$, where $s_0 \in S_0$. A state s_i is *reachable* if it belongs to some run r.

Example 1. We give in Fig. 2.1 a simple example of LTS containing two states s_0, s_1 (represented as circles), a transition from s_0 to s_1 (represented as an edge) labeled by an action a and a self-loop on s_1 labeled by an action b. The initial state is s_0.

2.3 Clocks, parameters and constraints

Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ denote the sets of non-negative integers, integers, non-negative rational numbers and nonnegative real numbers respectively.

2.3.1 Clocks

We assume a set $X = \{x_1, \ldots, x_H\}$ of *clocks*, i. e. real-valued variables that evolve at the same rate. A clock valuation is a function $w : X \to \mathbb{R}_+$. We identify a clock valuation w with the point $(w(x_1), \ldots, w(x_H))$ of $\mathbb{R}_+^H$. We write $\vec{0}$ for the clock valuation that assigns 0 to all clocks. Given $d \in \mathbb{R}_+$, $w + d$ denotes the valuation such that $(w + d)(x) = w(x) + d$, for all $x \in X$. Given $R \subseteq X$, we define the *reset* of a valuation w, denoted by $[w]_R$, as follows: $[w]_R(x) = 0$ if $x \in R$, and $[w]_R(x) = w(x)$ otherwise.

2.3.2 Parameters

We assume a set $P = \{p_1, \ldots, p_M\}$ of *parameters*, i. e. unknown constants. A parameter *valuation* v is a function $v : P \to \mathbb{Q}_+$. We identify a valuation v with the point $(v(p_1), \ldots, v(p_M))$ of $\mathbb{Q}_+^M$. An *integer* parameter valuation is a valuation v such that $\forall p \in P, v(p) \in \mathbb{N}$.

2.3.3 Constraints

We assume $\triangleleft \in \{<, \leq\}$ and $\bowtie \in \{<, \leq, \geq, >\}$.

Linear terms Linear term lt denotes a linear term over $X \cup P$ of the form $\sum_{1 \leq i \leq H} \alpha_i x_i + \sum_{1 \leq j \leq M} \beta_j p_j + d$, with $x_i \in X$, $p_j \in P$, and $\alpha_i, \beta_j, d \in \mathbb{Z}$.

Similarly, plt denotes a parametric linear term over P, that is a linear term without clocks ($\alpha_i = 0$ for all i).

A *constraint* C (i. e. a convex polyhedron) over $X \cup P$ is a set of inequalities of the form $lt \bowtie lt'$, with lt, lt' two linear terms.

We denote by *true* (resp. *false*) the constraint that corresponds to the set of all possible (resp. the empty set of) valuations.

Guards A *guard* g is a constraint over $X \cup P$ defined by inequalities of the form $x \bowtie plt$. We assume w.l.o.g. that, in each guard, given a clock x, at most one inequality is in the form $x \triangleleft plt$, that is a clock has a single upper bound (or none). A non-parametric guard is a guard over X, i. e. with inequalities $x \bowtie z$, with $z \in \mathbb{N}$.

Parametric zones A *parametric zone* C is a constraint over $X \cup P$ defined by inequalities of the form $x_i - x_j \bowtie plt$.

Parametric constraints A *parametric constraint* K is a constraint over P defined by inequalities of the form $plt \bowtie plt'$, with plt, plt' two parametric linear terms. We use the notation $v \models K$ to indicate that valuating parameters p with $v(p)$ in K evaluates to true. We denote by $\top$ (resp. $\bot$) the parametric constraint that corresponds to the set of all possible (resp. the empty set of) parameter valuations. Given two parametric constraints K_1 and K_2, we write $K_1 \subseteq K_2$ whenever for all v, $v \models K_1 \Rightarrow v \models K_2$.

Operations on constraints We denote by $C{\downarrow_P}$ the *projection* of C onto P, obtained by eliminating the clock variables.

We define the *time elapsing* of C, denoted by $C^{\nearrow}$, as the constraint over X and P obtained from C by delaying an arbitrary amount of time.

Given $R \subseteq X$, we define the *reset* of C, denoted by $[C]_R$, as the constraint obtained from C by resetting the clocks in R, and keeping the other clocks unchanged.

Given a parameter valuation v, $C[v]$ denotes the constraint over X obtained by replacing each parameter p in C with $v(p)$. Likewise, given a clock valuation w, $C[v][w]$ denotes the expression obtained by replacing each clock x in $C[v]$ with $w(x)$. We say that v *satisfies* C, denoted by $v \models C$, if the set of clock valuations satisfying $C[v]$ is nonempty. Given a parameter valuation v and a clock valuation w, we denote by $<w|v>$ the valuation over $X \cup P$ such that for all clocks x, $x[<w|v>] = x[w]$ and for all parameters p, $p[<w|v>] = p[v]$. We

use the notation $<w|v> \models C$ to indicate that $C[v][w]$ evaluates to true. We say that C is *satisfiable* if $\exists w, v$ s.t. $< w|v> \models C$.

2.4 Timed Automata

2.4.1 Introduction

Timed automata (TAs) [AD90, AD94], are an extension of finite state automata and a sub-class of hybrid automata [Hen96]. This is a widely used formalism for real-time systems modeling and verification with timing constraints, providing explicit manipulation of clock variables.

In a timed automaton, real-time behavior is captured by clock constraints on system transitions, setting or resetting clocks, etc. All real-valued clocks increase at the same rate. Clock values can be compared with constants in constraints called “invariants” that allow (or not) to stay in a location (sets of linear inequalities that must be satisfied to be able to remain in a location) or in constraint called “guards” to take a transition (sets of linear inequalities that must be satisfied to be able to take a transition). Moreover, some of the clocks can be reset when executing a transition.

Parallel composition of several timed automata is defined as a single timed automaton. It provides the designer with a powerful and intuitive way to represent timed systems. It is important to note that the verification of the timed automaton is very sensitive to the size of each automaton and the number of automata in parallel, thus often leading to the state space explosion problem.

2.4.2 Syntax

Definition 2.4.1. A *timed automaton* (TA) $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, l_0, F, X, I, E)$, where: *i)* Σ is a finite set of actions, *ii)* L is a finite set of locations, *iii)* $l_0 \in L$ is the initial location, *iv)* F is a set of final locations, *v)* X is a set of clocks, *vi)* I is the invariant function, assigning to every location $l \in L$ a constraint $I(l)$, *vii)* E is a set of edges $e = (l, g, a, R, l')$ where $l, l' \in L$ are the source and target locations, $a \in \Sigma$ is an action, $R \subseteq X$ is a set of clocks to be reset, and g is a guard.

In practice, discrete variables are often used in TA, located on transitions. The value of discrete variable can be used in guards and updated by the transitions. Note that we will not introduce them in any theoretical part of this thesis but we will use them in our system models.

We give here conventions for the graphical representation of TA: locations are represented by nodes, next to which the invariant of the location is written; transitions are represented by arcs going from one location to another location or the same location (self-loops). Next to a transition are the associated guard, the action name and the set of clocks to be reset. Note that constraints in guards and invariants equal to true will be omitted. The initial location is usually represented with an unlabeled arrow pointing from nothing to it.

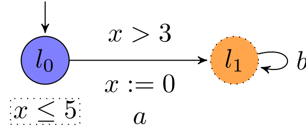


Figure 2.2 – Example of timed automaton

Example 2. We give in Fig. 2.2 a simple example of TA containing two locations l_0 and l_1, an action a and a clock x. The initial location is l_0.

In this TA, location l_0 has invariant $x \leq 5$, location l_1 has invariant *true*. The transition from location l_0 to location l_1 has guard $x > 3$ through action a and resets clock x.

In the beginning, the system stays at location l_0 with invariant $x \leq 5$. Then it can increase an amount of time less than 5 and take the transition having the action a to go to location l_1 when $x > 3$. The clock x will be reset to zero by the transition before entering location l_1. After that, the system can take the self-loop at any time or the clock x can increase infinitely at location l_1.

Parallel composition of TAs Several TAs can cooperate. This is obtained by building a product of TAs. A state in the product is a pair of states, where each state indicates the progress one of the TA has made. A transition in the product can be a local transition of a TA or a synchronization between several TAs. We present here the definition and notion of network of timed automata and show how N TAs can be composed into a single TA. Note that, for simplicity purposes, shared clock variables will not be used in our theoretical example.

Definition 2.4.2 (synchronized product of TAs)**.** Let $N \in \mathbb{N}$. Given a set of TAs $\mathcal{A}_i = (\Sigma_i, L_i, (l_0)_i, F_i, X_i, I_i, E_i)$, $1 \leq i \leq N$, the *synchronized product* of $\mathcal{A}_i$, $1 \leq i \leq N$, denoted by $\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \cdots \parallel \mathcal{A}_N$, is the tuple $(\Sigma, L, l_0, F, X, I, E)$, where:

1. $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,
2. $L = \prod_{i=1}^{N} L_i$,
3. $l_0 = ((l_0)_1, \ldots, (l_0)_N)$,
4. $F = \{(l_1, \ldots, l_N) \in L \mid \exists i, 1 \leq i \leq N \text{ s.t. } l_i \in F_i\}$,
5. $X = \bigcup_{1 \leq i \leq N} X_i$,
6. $I((l_1, \ldots, l_N)) = \bigwedge_{i=1}^{N} I_i(l_i)$ for all $(l_1, \ldots, l_N) \in L$,

and E is defined as follows. For all $a \in \Sigma$, let ζ_a be the subset of indices $i \in 1, \ldots, N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $(l_1, \ldots, l_N) \in L$, for all $(l'_1, \ldots, l'_N) \in L$, $\big((l_1, \ldots, l_N), g, a, R, (l'_1, \ldots, l'_N)\big) \in E$ iff:

- for all $i \in \zeta_a$, there exist g_i, R_i such that $(l_i, g_i, a, R_i, l'_i) \in E_i$, $g = \bigwedge_{i \in \zeta_a} g_i$, $R = \bigcup_{i \in \zeta_a} R_i$, and,
- for all $i \notin \zeta_a$, $l'_i = l_i$.

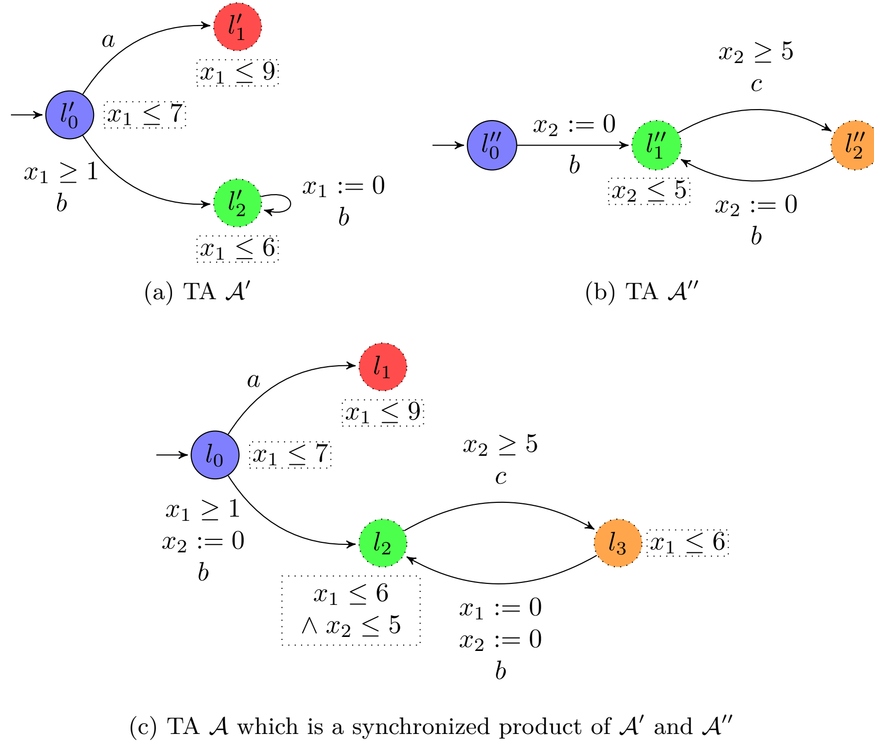
Example 3. We give in Fig. 2.3 an example of a network of two TAs $\mathcal{A}'$ and TA $\mathcal{A}''$, presented in Fig. 2.3a and Fig. 2.3b respectively. The synchronized product of these 2 PTAs $\mathcal{A}' \parallel \mathcal{A}''$ corresponds to the TA in Fig. 2.3c, where $l_0 = (l'_0, l''_0)$, $l_1 = (l'_1, l''_0)$, $l_2 = (l'_2, l''_1)$ and $l_3 = (l'_2, l''_2)$.

2.4.3 Concrete semantics

The semantics of TAs is constructed from a LTS, where states are made by a location and a valuation for each clock. Given a TA $\mathcal{A} = (\Sigma, L, l_0, F, X, I, E)$:

Concrete state A concrete state of $\mathcal{A}$ is a pair $\mathbf{s} = (l, w)$ where $l \in L$ is a location, and w is a valuation of each clock.

Initial concrete state The initial concrete state of $\mathcal{A}$ is $\mathbf{s}_0 = \big(l_0, \vec{0}\big) = \big(l_0, 0\big)$. That is, the initial state corresponds to all clocks equal to 0.



(a) TA $\mathcal{A}'$

(b) TA $\mathcal{A}''$

(c) TA $\mathcal{A}$ which is a synchronized product of $\mathcal{A}'$ and $\mathcal{A}''$

Figure 2.3 – Example of network of parametric timed automata

Definition 2.4.3 (Concrete semantics of a TA)**.** The concrete semantics of $\mathcal{A}$ is given by the LTS $(S, s_0, \rightarrow)$, with $S = \{(l, w) \in L \times \mathbb{R}_+^H \mid I(l)[w] \text{ is true}\}$, $s_0 = (l_0, \vec{0})$, and $\rightarrow$ consists of the discrete and (continuous) delay transition relations:

- discrete transitions: $(l, w) \xrightarrow{e} (l', w')$, if $(l, w), (l', w') \in S$, there exists $e = (l, g, a, R, l') \in E$, $w' = [w]_R$, and $g[w]$ is true.

- delay transitions: $(l, w) \xrightarrow{d} (l, w + d)$, with $d \in \mathbb{R}_+$, if $\forall d' \in [0, d], (l, w + d') \in S$.

We write $(l, w) \xrightarrow{d,e} (l', w')$ for a sequence of delay and discrete transitions where $((l, w), e, (l', w')) \in \rightarrow$ if $\exists d, w'' : (l, w) \xrightarrow{d} (l, w'') \xrightarrow{e} (l', w')$.

Concrete run A *concrete run* is a sequence $r = s_0\alpha_0 s_1\alpha_1 \cdots s_n\alpha_n \cdots$ s.t. $\forall i, (s_i, \alpha_i, s_{i+1}) \in \rightarrow$. We consider as usual that concrete runs strictly alternate delays d_i and discrete transitions e_i and we thus write concrete runs in the form $r = s_0 \xrightarrow{(d_0,e_0)} s_1 \xrightarrow{(d_1,e_1)} \cdots$. We refer to a state of a run starting from the initial state of a TA $\mathcal{A}$ as a *concrete state* of $\mathcal{A}$. Note that when a run is finite, it must

end with a concrete state.

An infinite run is said to be *Zeno* if it contains an infinite number of discrete transitions within a finite delay, i. e. if the sum of all delays d_i is bounded.

Example 4. Fig. 2.4 depicts an example of concrete run of the TA $\mathcal{A}$ in Fig. 2.3c. This run starts at the initial location l_0 on which the values of both clocks equal zero. Then they both increase by 1 time units and reset x_2 to zero before taking the action b. After that, in l_2 we spend 5 time units and then take action c to go to l_3. Continuously, at l_3 we take action c and reset values of both clocks immediately to go back to l_2. Again, we spend at least 5 time units in l_3 before taking action c to go back to l_2, and so on.

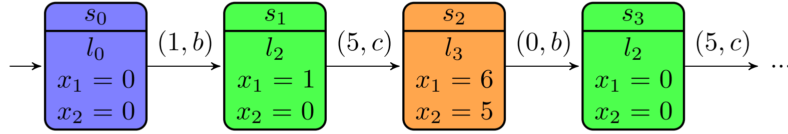


Figure 2.4 – Example of concrete run for the TA $\mathcal{A}$ in Fig. 2.3c

Trace Let $(l_0, w_0) \overset{(d_0,e_0)}{\rightarrow} (l_1, w_1) \overset{(d_1,e_1)}{\rightarrow} \cdots \overset{(d_{m-1},e_{m-1})}{\rightarrow} (l_m, w_m)$ be a run of $\mathcal{A}$, its corresponding *trace* is $l_0 \overset{e_0}{\Rightarrow} l_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} l_m$. In fact, the trace is built from a run by removing the valuation of the clocks. It can be seen as a "time abstract" run. Two concrete runs are said to be equivalent if their corresponding traces are equal.

Example 5. The trace associated with the concrete run of Fig. 2.4 is depicted in Fig. 2.5.

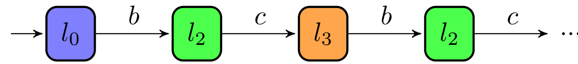


Figure 2.5 – Example of trace for the TA $\mathcal{A}$ in Fig. 2.3c

Trace set The trace set of $\mathcal{A}$ is the set of traces associated with all maximal runs of $\mathcal{A}$.

Example 6. The trace set associated with the TA $\mathcal{A}$ in Fig. 2.3c is depicted in Fig. 2.6.

2.4.4 Problems

Although TAs were successfully used for verifying models of complex distributed systems using powerful model checkers, they still suffer from some limitations:

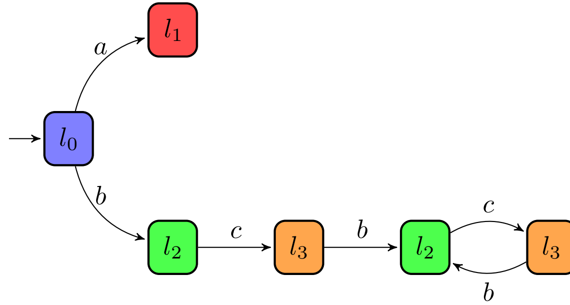


Figure 2.6 – Example of trace set for the TA $\mathcal{A}$ in Fig. 2.3c

1. TA cannot model systems incompletely specified (i. e. timing constants are not known yet). Therefore, with such systems, it cannot be used at the beginning of design phases.

2. Verifying a system for a set of timing constants usually requires replacing all of them one by one if they are supposed to be integer-valued. And it is infeasible if they are real-valued (dense interval).

3. Robustness in TA often assumes that all guards have delays that can be enlarged or shrunk by the same small variation. But independent variations or considering both enlarging and shrinking was not addressed, and it is actually unclear whether this can be even considered for TA although the robust semantics can help tackling some problems, see e. g. [Mar11].

In the next section we will introduce parametric timed automaton (PTA) [AHV93] which can leverage these drawbacks by allowing the use of timing parameters, hence allowing for modeling constants unknown or known with some imprecision.

2.4.5 Tools and applications

Tools TAs have been studied in various settings (such as planning [KMH01]) and benefit from powerful model checkers such as Uppaal [LPY97], Kronos [BDM+98], PAT [SLDP09], Roméo [LRST09], TReX [ABS01] or the schedulability analyser TIMES [AFM+02], etc.

Applications Over the past two decades, TA based tools have become more and more mature, some academic research tools now being used in various industries. These verification tools for TA based models such as Uppaal, Kronos, PAT and Rabbit have also proven to be most successful [LPY97, BDM+98, Wan02, BLN03], and have been used to model and verify a variety of hardware systems and software systems such as network and telecommunication protocols, hardware circuits, distributed algorithms, security protocols, web service protocols, etc.

2.5 Parametric timed automata

2.5.1 Introduction

Parametric timed automata (PTAs) are an extension of timed automata to the parametric case. It allows, within guards and invariants, the use of parameters (i. e. unknown constants) in place of constants [AHV93]. By adding parameters, the model checking problem with a binary answer ("yes/no") becomes the *parameter synthesis* problem with a richer answer: a set of valuations for which a property holds. This parametric model is interesting when one does not only want to check that a system is correct for one value of the constants, but for a whole dense set of values. Besides, the parametric timed automata model is also interesting to synthesize parameters for which a given property is satisfied.

2.5.2 Syntax

Definition 2.5.1. A *parametric timed automaton* (PTA) $\mathcal{A}$ is a 9-tuple $\mathcal{A} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$, where: *i)* Σ is a finite set of actions, *ii)* L is a finite set of locations, *iii)* $l_0 \in L$ is the initial location, *iv)* F is a set of final locations, *v)* X is a set of clocks, *vi)* P is a set of parameters, *vii)* K_0 is the initial parameter constraint, *viii)* I is the invariant function, assigning to every location $l \in L$ a constraint $I(l)$, *ix)* E is a set of edges $e = (l, g, a, R, l')$ where $l, l' \in L$ are the source and target locations, $a \in \Sigma$ is an action, $R \subseteq X$ is a set of clocks to be reset, and g is a guard.

Similarly to the PTAs, we provide below the basic definitions of Parametric Timed Büchi Automata, a variant of PTAs with replacing the set of final locations with a set of accepting locations.

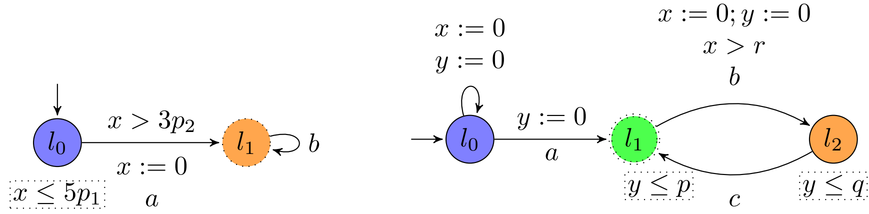
Definition 2.5.2. A *Parametric Timed Büchi Automaton* (PTBA) $\mathcal{B}$ is a 9-tuple $\mathcal{B} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$, where: *i)* Σ is a finite set of actions, *ii)* L is a finite set of locations, *iii)* $l_0 \in L$ is the initial location, *iv)* F is a set of *accepting locations*, *v)* X is a set of clocks, *vi)* P is a set of parameters, *vii)* K_0 is the initial parameter constraint, *viii)* I is the invariant function, assigning to every location $l \in L$ a constraint $I(l)$, *ix)* E is a set of edges $e = (l, g, a, R, l')$ where $l, l' \in L$ are the source and target locations, $a \in \Sigma$ is an action, $R \subseteq X$ is a set of clocks to be reset, and g is a guard.

Given a PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, K_0, I, E)$, and a parameter valuation v, $\mathcal{A}[v]$ denotes the TA obtained from $\mathcal{A}$ by substituting every occurrence of a parameter p_i by the constant $v(p_i)$ in the guards and invariants.

The initial constraint K_0 is used to constrain some parameters (as in, e. g. [HRSV02, ACEF09]). In other words, it defines a domain of valuation for the parameters. For example, given two parameters $p_{\min}$ and $p_{\max}$, we may want to ensure that $p_{\min} \leq p_{\max}$. Given $\mathcal{A} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$, we write $\mathcal{A}.K_0$ as a shortcut for the initial constraint of $\mathcal{A}$. In addition, given K_0', we denote by $\mathcal{A}(K_0')$ the PTA where $\mathcal{A}.K_0$ is replaced with K_0'.

Given a parameter valuation $v \models \mathcal{A}.K_0$, we denote by $\mathcal{A}[v]$ the non-parametric TA where all occurrences of a parameter p_i have been replaced by $v(p_i)$. If $v \not\models \mathcal{A}.K_0$, we assume the model is not defined (i. e. it corresponds to the empty TA, with no location).

Example 7. We give in Fig. 2.7a an example of PTA containing two locations l_0 and l_1, an action a, 2 parameters p_1 and p_2, and a clock x. The initial location is l_0. In this PTA, location l_0 has invariant $x \leq 5p_1$, location l_1 has invariant *true*. The transition from location l_0 to location l_1 has guard $x > 3p_2$ through action a and resets clock x. Thus, it can increase an amount of time less than $5p_1$ and take the transition having the action a to go to location l_1 as soon as $x > 3p_2$. The clock x will be reset to zero. After that, the system can take the self-loop at any time or the clock x can increase infinitely at location l_1.



(a) A parametric timed automaton (b) A parametric timed Büchi automaton

Figure 2.7 – Examples of parametric timed automaton and parametric timed Büchi automaton

Example 8. Fig. 2.7b shows a PTBA, where only location l_1 is accepting. It has two clocks x and y, and three parameters p, q and r used in the guard of action b and in the invariants of locations l_1 and l_2.

Parallel composition of PTAs Similarly to the parallel composition of TAs, we now introduce the notion of a network of parametric timed automata, and show in the following definition how N PTAs can be composed into a single PTA.

Definition 2.5.3 (synchronized product of PTAs)**.** Let $N \in \mathbb{N}$. Given a set of PTAs $\mathcal{A}_i = (\Sigma_i, L_i, (l_0)_i, F_i, X_i, P_i, (K_0)_i, I_i, E_i)$, $1 \leq i \leq N$, the *synchronized product* of $\mathcal{A}_i$, $1 \leq i \leq N$, denoted by $\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \cdots \parallel \mathcal{A}_N$, is the tuple $(\Sigma, L, l_0, F, X, P, K_0, I, E)$, where:

1. $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,
2. $L = \prod_{i=1}^{N} L_i$,
3. $l_0 = ((l_0)_1, \dots, (l_0)_N)$,
4. $F = \{(l_1, \dots, l_N) \in L \mid \exists i, 1 \leq i \leq N \text{ s.t. } l_i \in F_i\}$,
5. $X = \bigcup_{1 \leq i \leq N} X_i$,
6. $P = \bigcup_{1 \leq i \leq N} P_i$,
7. $K_0 = \bigwedge_{i=1}^{N} (K_0)_i$
8. $I((l_1, \dots, l_N)) = \bigwedge_{i=1}^{N} I_i(l_i)$ for all $(l_1, \dots, l_N) \in L$,

and E is defined as follows. For all $a \in \Sigma$, let ζ_a be the subset of indices $i \in 1, \dots, N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $(l_1, \dots, l_N) \in L$, for all $(l'_1, \dots, l'_N) \in L$, $\big((l_1, \dots, l_N), g, a, R, (l'_1, \dots, l'_N)\big) \in E$ iff:

- for all $i \in \zeta_a$, there exist g_i, R_i such that $(l_i, g_i, a, R_i, l'_i) \in E_i$, $g = \bigwedge_{i \in \zeta_a} g_i$, $R = \bigcup_{i \in \zeta_a} R_i$, and,
- for all $i \notin \zeta_a$, $l'_i = l_i$.

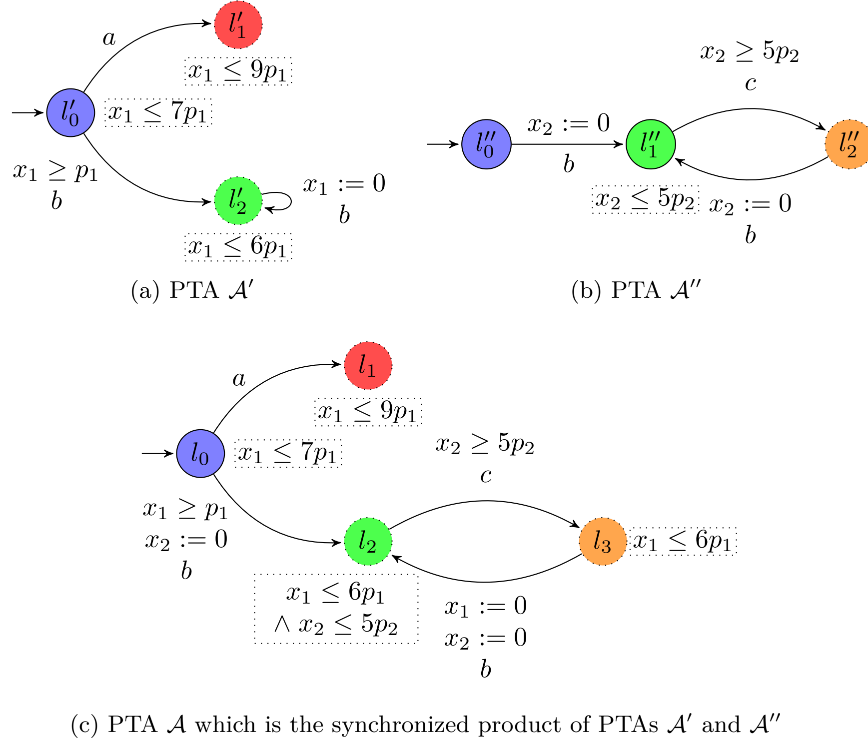
Example 9. We give in Fig. 2.8 an example of network of two PTAs, $\mathcal{A}'$ and $\mathcal{A}''$, presented in Fig. 2.8a and Fig. 2.8b respectively. The synchronized product of these 2 PTAs $\mathcal{A}' \parallel \mathcal{A}''$ corresponds to the PTA in Fig. 2.8c, where $l_0 = (l'_0, l''_0)$, $l_1 = (l'_1, l''_0)$, $l_2 = (l'_2, l''_1)$ and $l_3 = (l'_2, l''_2)$.

2.5.3 Concrete semantics

The semantics of PTAs is constructed from a LTS, where states are made by a location and a valuation for each clock. Let us recall the concrete semantics of PTA (as in e. g. [JLR15]). Given a PTA $\mathcal{A} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$ and a parameter valuation v. By substituting parameters with the parameter valuation v, the PTA $\mathcal{A}$ becomes TA $\mathcal{A}[v]$ and has the same concrete semantics as TA in Section 2.4.3.

2.5.4 Symbolic semantics

Instead of representing each individual state separately as in concrete semantics of PTAs (Section 2.5.3), symbolic semantics of PTAs can represent and manipulate sets of concrete states.



(a) PTA $\mathcal{A}'$

(b) PTA $\mathcal{A}''$

(c) PTA $\mathcal{A}$ which is the synchronized product of PTAs $\mathcal{A}'$ and $\mathcal{A}''$

Figure 2.8 – Example of network of parametric timed automata

Let us recall the symbolic semantics of PTA (as in e. g. [ACEF09]). Given a PTA $\mathcal{A} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$:

Symbolic state A symbolic state is a pair $\mathbf{s} = (l, C)$ with l a location, and C a constraint over $X \cup P$ (or *zone*). We may view a symbolic state s as the set of pairs (l, w) where w is a clock valuation such that there exists a parameter valuation v such that $< w, v > \models C$.

Initial symbolic state The initial state of $\mathcal{A}$ is $\mathbf{s}_0 = (l_0, (X = 0)^{\nearrow} \wedge I(l_0))$, i. e. clocks are initially set to 0, and can evolve as long as $I(l_0)$ is satisfied.

Initial constraint The constraint K_0 corresponds to the initial constraint over the parameters P. For example, in a PTA with two parameters p_1 and p_2, we may want to constrain p_1 to be always smaller or equal to p_2, or p_1 greater than p_2, in which case K_0 is defined as $K_0 = p_1 \leq p_2$ or $K_0 = p_1 > p_2$.

Computation of the state space The computation of the state space relies on the Succ operation as follows: Given a symbolic state $\mathbf{s} = (l, C)$,

$\mathsf{Succ}(\mathbf{s}) = \{(l', C') \mid \exists(l, g, a, R, l') \in E \text{ s.t. } C' = \big([(C \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')\}$. The Succ operation is effectively computable, using polyhedra operations: note that the successor of a parametric zone C is a parametric zone (see e. g. [JLR15]).

Definition 2.5.4 (Symbolic semantics of a PTA)**.** The symbolic semantics of $\mathcal{A}$ is the LTS called parametric zone graph $\mathsf{PZG} = (E, \mathbf{S}, \mathbf{s}_0, \Rightarrow)$, with $\mathbf{s} \in S$, $\mathbf{s} = \{(l, C) \mid C \subseteq I(l)\}$, $\big((l, C), e, (l', C')\big) \in \Rightarrow$ if $e = (l, g, a, R, l')$ and $C' = \big([(C \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')$ with C' satisfiable. Note that if $\big((l, C), e, (l', C')\big) \in \Rightarrow$, we can write $\mathsf{Succ}(\mathbf{s}, e) = (l', C')$.

The *parametric zone graph* $\mathsf{PZG}(\mathcal{A})$ of a PTA $\mathcal{A}$ is made of the states of $\mathcal{A}$, and there is an edge in $\mathsf{PZG}(\mathcal{A})$ from $\mathbf{s}_i$ to $\mathbf{s}_j$ whenever $\mathbf{s}_j \in \mathsf{Succ}(\mathbf{s}_i)$. In the PZG, nodes are symbolic states, and arcs are labeled by *edges* S of the original PTA.

Similar to the case in timed automata, the parametric zone graph generated from a PTA $\mathcal{A}$ may be infinite. In (plain) timed automata, extrapolations are used to obtain finite zone graphs. For instance, k- or lu-extrapolation uses the extremal constant values appearing as lower- or upper-bounds in clock constraints of the TA, in order to identify symbolic zones that cannot be distinguished. Indeed, it is not obvious how extrapolation carries over in case clock constraints have parameters. Therefore, we are forced to accept that parametric zone graphs can be infinite.

Symbolic run A symbolic run of a PTA is an alternating sequence of symbolic states and edges of the form $\mathbf{s}_0 \overset{e_0}{\Rightarrow} \mathbf{s}_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} \mathbf{s}_m$, such that for all $i = 0, \dots, m-1$, $e_i \in E$, and $\mathbf{s}_i \overset{e_i}{\Rightarrow} \mathbf{s}_{i+1}$ is such that $\mathbf{s}_{i+1}$ belongs to $\mathsf{Succ}(\mathbf{s}_i)$ and is obtained via edge e_i. In the following, we simply refer to the symbolic states belonging to a run of $\mathcal{A}$ starting from $\mathbf{s}_0$ as states of $\mathcal{A}$.

Example 10. By using the operation Succ presented previously for constructing a symbolic run of PTA $\mathcal{A}$, Fig. 2.9 depicts an example of symbolic run of the PTA $\mathcal{A}$ in Fig. 2.8c.

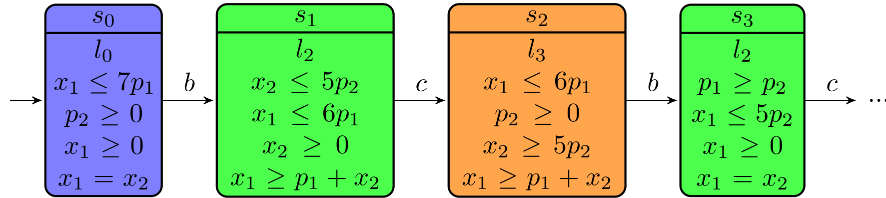


Figure 2.9 – Example of symbolic run for the PTA $\mathcal{A}$ in Fig. 2.8c

Example 11. The PZG example of the PTA $\mathcal{A}$ in Fig. 2.8c and PBTA $\mathcal{B}$ in Fig. 2.7b depicted in Fig. 2.10 and Fig. 2.11 respectively, and they are also constructed by using the operation Succ.

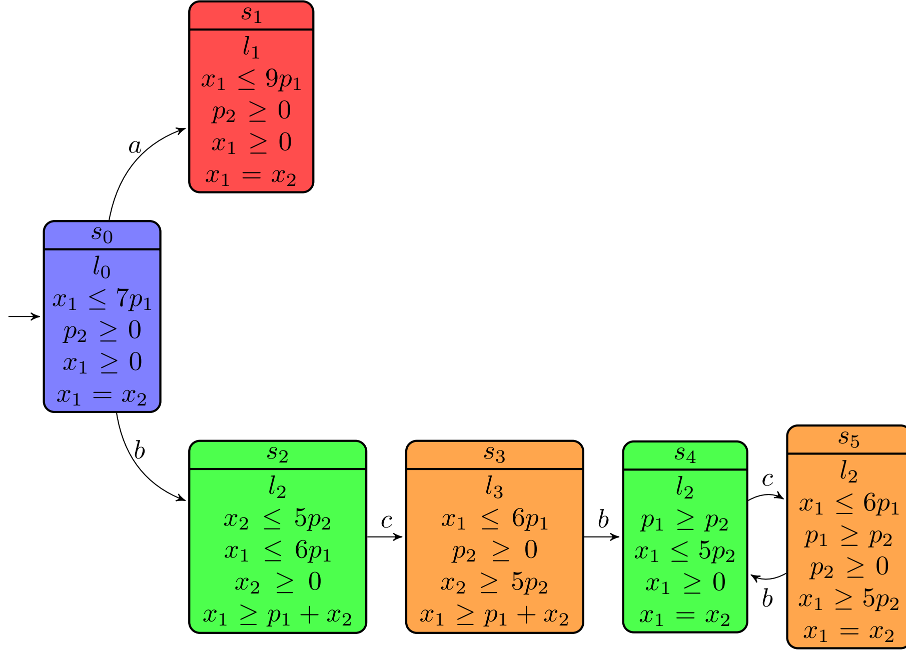


Figure 2.10 – Example of parametric zone graph PZG for the PTA $\mathcal{A}$ in Fig. 2.8c

Trace Given a run $(l_0, C_0) \overset{e_0}{\Rightarrow} (l_1, C_1) \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} (l_m, C_m)$, its corresponding *trace* is $l_0 \overset{e_0}{\Rightarrow} l_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} l_m$. In fact, the trace is built from a run by removing the valuation of the clocks and parameters, and thus can be seen as a "time abstract" run. Two runs (concrete or symbolic) are said to be equivalent if their associated traces are equal.

Example 12. The trace associated with the symbolic run of Fig. 2.9 is depicted in Fig. 2.12.

Trace set The set of all traces of a PTA is called its *trace set*.

Example 13. The trace set associated with the PTA $\mathcal{A}$ in Fig. 2.8c is depicted in Fig. 2.13.

2.5.5 Subsumption abstraction

We now discuss how to carry over subsumption abstraction from timed automata to the parametric case. For timed automata, subsumption was introduced in [DT98] and studied in [ELPvdP12] in the context of LTL model checking. The idea is that a symbolic state may be replaced by a "larger" one, without losing behaviour.

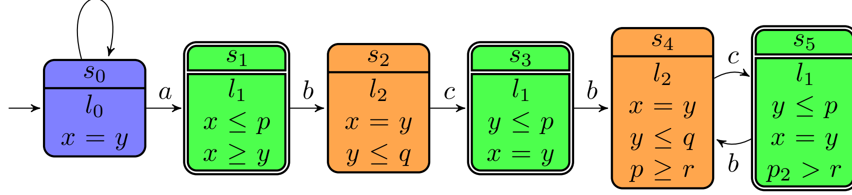


Figure 2.11 – Example of parametric zone graph PZG for the PBTA in Fig. 2.7b

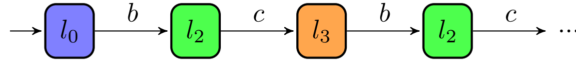


Figure 2.12 – Example of trace for the PTA $\mathcal{A}$ in Fig. 2.8c

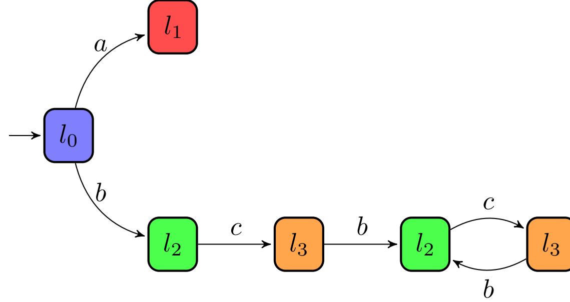


Figure 2.13 – Example of trace set for the PTA $\mathcal{A}$ in Fig. 2.8c

Definition 2.5.5 (Subsumption $\sqsubseteq$)**.** A state $\mathbf{s} = (l, C) \in \mathbf{S}$ is subsumed by another $\mathbf{s}' = (l', C')$, denoted $\mathbf{s} \sqsubseteq \mathbf{s}'$, when $l = l'$ and $C \subseteq C'$.

Definition 2.5.6 (Subsumption Abstraction)**.** An *abstraction* over the Parametric Zone Graph $\mathsf{PZG}(\mathcal{A}) = (\mathbf{S}, \mathbf{s}_0, \Rightarrow)$ is a total mapping $\alpha : \mathbf{S} \to \mathbf{S}$ s.t. for all reachable symbolic states $\mathbf{s}$, we have $\mathbf{s} \sqsubseteq \alpha(\mathbf{s})$.

Definition 2.5.7 (Induced PZG)**.** An abstraction α over the Parametric Zone Graph $\mathsf{PZG}(\mathcal{A}) = (\mathbf{S}, \mathbf{s}_0, \Rightarrow)$ induces an abstracted Parametric Zone Graph $\mathsf{PZG}_\alpha(\mathcal{A}) = (\mathbf{S}_\alpha, \alpha(\mathbf{s}_0), \Rightarrow_\alpha)$, where:

- $\mathbf{S}_\alpha = \{\alpha(\mathbf{s}) \mid \mathbf{s} \in \mathbf{S}\}$ is the set of states, s.t. $\mathbf{S}_\alpha \subseteq \mathbf{S}$,
- $\alpha(\mathbf{s}_0)$ is the initial state, and
- the transition relation is: $\mathbf{s} \Rightarrow_\alpha \mathbf{s}'$ iff there exists $\mathbf{s}''$ s.t. $\mathbf{s} \Rightarrow \mathbf{s}''$ and $\mathbf{s}' = \alpha(\mathbf{s}'')$.

Note that if the image of the abstraction is finite, the abstract PZG is finite as well. However, this is not always the case. The following lemma shows that

indeed abstraction does not lose behaviour.

Proposition 2.5.1 ($\sqsubseteq$ is a simulation relation)**.** *If* $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2$ *and* $\mathbf{s}_1 \Rightarrow \mathbf{s}_1'$ *then there exists* $\mathbf{s}_2'$ *s.t.* $\mathbf{s}_2 \Rightarrow \mathbf{s}_2'$ *and* $\mathbf{s}_1' \sqsubseteq \mathbf{s}_2'$*.*

Proof. By the definition of $\sqsubseteq$, and the fact that the symbolic transition relation $\Rightarrow$ is monotone w.r.t. $\subseteq$ of zones. □

Note that in practice, the subsumption abstraction is defined only over the reachable state space, and it might introduce extra behaviour that the unabstracted system cannot simulate. Typically α is constructed on-the-fly, i. e. during analysis, by only abstracting to states that have already been found to be reachable. This makes its performance depend heavily on the search order. In particular, finding "large" states as early as possible can make the abstraction coarser [DLL+12].

2.5.6 Problems

The main problem of PTA formalism is decidability. Unfortunately, unlike the decidability of TA [BY03], almost all interesting parameter synthesis problems for PTA [AHV93] are known to be undecidable in general, including the emptiness of the valuations set for which a given location is reachable. We will dedicate Section 2.8 to decidability, and Section 2.8.1 to parameter synthesis problems which we will use in this thesis.

2.5.7 Tools and applications

Tools To the best of our knowledge, IMITATOR [AFKS12] for parametric stopwatch automata is the only tool that takes PTA as an input.

Applications Beyond academic examples (such as variants of train controllers [AHV93, HRSV02]), PTAs have been successfully used to specify and verify numerous interesting case studies such as: e. g. the root contention protocol [HRSV02], Philip's bounded retransmission protocol [HRSV02], a 4-phase handshake protocol [KP12], the alternating bit protocol [JLR15], an asynchronous circuit commercialized by STMicroelectronics [CETFX09], (non-preemptive) schedulability problems [JLR15, CPR08], a distributed prospective architecture for the flight control system of the next generation of spacecrafts designed at ASTRIUM Space Transportation [FSLM12], an unmanned aerial video system by Thales, and even analysis of music scores [FJ13].

2.5.8 Related formalisms

Timed automata are not the only formalism extended to the parametric case. Other popular formalisms are also extended to the parametric case, to model and verify real-time systems such as:

- **Hybrid Automata with Parameters (HAs)** [AHV93, HH94]: State space exploration algorithms for hybrid automata have been extended to allow a parametric analysis and implemented in the tool Hytech [HHW97] or HYMITATOR. .

- **Parametric Interrupt Timed Automata (PITAs)** [BHJL13]: A parametrized version of interrupt timed automata, where polynomials of parameters can occur in guards and updates. Different reachability problems, including robust reachability, are were proven to be decidable for this model.

- **Parametric Timed Petri Nets (PTPNs)** [TLR09]: Timed Petri Nets (TPNs) are also a well-known formalism to model real-time systems besides Timed Automata (TA), and parametric timed Petri nets are its parametric extension.

For extensions of PTA, there are e. g. action synthesis problem for the parameters [KMP15, AKPP16], probabilistic parameters [AFS13b].

2.6 System property specification

Given a model of a system and a specification of its desired properties [BK08], a model-checker will check their satisfaction automatically and exhaustively. A diversity of properties can be verified by model-checking, such as deadlock freedom, invariants, request-response properties, etc. However, these properties can be categorized into five classes: reachability, safety, liveness, deadlock-freeness, and fairness properties.

- *Reachability property*: A certain state of system *can be reached*.

- *Safety property*: *Desirable states* of system should occur or *an undesirable state* of system should never occurs (deadlock-freeness).

- *Liveness property*: A certain state of system *will eventually occur*.

- *Deadlock-freeness property*: A special property expressing that the system can never be stuck and thus make no progress.

- *Fairness property*: A certain state of system will occur (or will fail to occur) *infinitely often*.

2.7 Temporal logic

Temporal logic [Pnu77, BK08] is an extension of propositional logic with temporal operators, which is dedicated for statements and reasoning that include the notion of order in time. In 1977, A. Pnueli first used it to specify behavioral properties (Linear Temporal Logic).

In the following, we present shortly the syntax of the *Linear Temporal Logic* (LTL) and the *Computation Tree Logic*, which are the two most basic and popular temporal logics to specify system properties including those in Section 2.6. Both of these temporal logics are widely used for various formalisms such as LTSs, TAs, Petri nets, Time Petri nets, Timed Petri nets, etc.

Linear Temporal Logic - LTL [Pnu77]. A *Linear Temporal Logic* is formula defined by:

$$\varphi ::= \bot | \top | ap | \neg\ \varphi | \varphi\ \wedge\ \varphi | \varphi\ \vee\ \varphi | \mathbf{G}\varphi | \mathbf{F}\varphi | \varphi\ \mathbf{U}\varphi | \mathbf{X}\varphi$$

Where *ap* is an atomic proposition, φ is a formula, $\mathbf{G}, \mathbf{F}, \mathbf{U}$ and $\mathbf{X}$ denote the *always*, *eventually*, *until* and *next* operators respectively.

Computation Tree Logic - CTL [CE81]. A *Computation Tree Logic* or branching-time logic is formula defined by:

$$\begin{gathered}\varphi ::= \bot | \top | a | \neg\ \varphi | \varphi\ \wedge\ \varphi | \varphi\ \vee\ \varphi |\ \mathbf{AG}\varphi | \mathbf{AF}\varphi | \mathbf{A}\varphi\mathbf{U}\varphi \\ | \mathbf{AX}\varphi | \mathbf{EG}\varphi | \mathbf{EF}\varphi | \mathbf{E}\varphi\mathbf{U}\varphi | \mathbf{EX}\varphi\end{gathered}$$

Where *ap* is an atomic proposition, φ is a formula, prefixes **A** and **E** denote *for all paths* and *there exists a path* respectively. $\mathbf{G}, \mathbf{F}, \mathbf{U}$ and $\mathbf{X}$ denote the *always*, *eventually*, *until* and *next* operators respectively.

The main difference between LTL and CTL is the quantifiers over paths, LTL semantics expresses a property of a single path. While the CTL semantics expresses a formula on all possible paths, considering either all possible paths (**A** operator) or only single path (**E** operator) when encountering a branch.

Furthermore, LTL and CTL are incomparable and are both subsets of CTL* [EH86]. There are also extensions of them and other temporal logics e. g. timed computational tree logic (TCTL) [ACD90], metric temporal logic (MTL) [Koy90], MITL [BL09], timed propositional temporal logic (TPTL) [AH89], temporal logic of actions (TLA) [Lam94], μ-calculus, etc.

Example 14. Let a system be given, and a good state "GoodState" and a bad state "BadState". The LTL formula **F**(GoodState) describes the reachability property that the system eventually reaches the GoodState. The CTL formula **AG**(¬BadState) expresses the safety property that the system never reaches the bad state in any case.

Note that, CTL notations such as $\mathbf{EF}, \mathbf{AF}, \mathbf{EG}$ and **AG** will be used repeatedly throughout this thesis.

2.8 State of the art

Model checking is known as being time-consuming. Indeed, to verify a large model of dependent sub-models, it usually consumes a lot of time to traverse

all possible states in order to verify such model satisfying all its specifications. Unfortunately, this procedure can even take forever and this problem was also known as an inevitable issue for parameter synthesis. Therefore, by proving a synthesis problem executed by the synthesis procedure is decidable, one can have the ability to predict whether the synthesis procedure will eventually terminate and then return complete results.

To convey an overall picture of the decidability problems of PTA, the first of this section is dedicated to addressing common parameter synthesis problems. Then the next parts show their related decidability results on PTA and its subclasses in the past few years.

2.8.1 Decision and computation problems

Due to the richer result of parameter synthesis, properties mentioned in Section 2.6 are no longer sufficient and should thus be extended and adapted.

Let $\mathcal{P}$ be a class of decision or computation problem (reachability, unavoidability, etc.) of parameter synthesis [JLR15]. We give in the following the general definitions of $\mathcal{P}$-emptiness, $\mathcal{P}$-universality, $\mathcal{P}$-finiteness and $\mathcal{P}$-synthesis problems:

Decision problems:

$\mathcal{P}$-emptiness problem:
INPUT: A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
PROBLEM: Is the set of parameter valuations v such that $\mathcal{A}[v]$ satisfies ϕ empty?

$\mathcal{P}$-universality problem:
INPUT: A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
PROBLEM: Are all parameter valuations v such that $\mathcal{A}[v]$ satisfies ϕ?

$\mathcal{P}$-finiteness problem:
INPUT: A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
PROBLEM: Is the set of parameter valuations v such that $\mathcal{A}[v]$ satisfies ϕ finite?

Computation problem:

$\mathcal{P}$-synthesis problem:
INPUT: A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
PROBLEM: Compute the parameter valuations v such that $\mathcal{A}[v]$ satisfies ϕ

Consider again the common reachability and unavoidability properties such as EF, AF, EG, AG, etc. expressed in CTL (see Section 2.7), and the liveness properties from [AL17] such as deadlock-existence ED, cycle-existence EC. The problem class $\mathcal{P}$ will be replaced with one of these properties to have a particular problem of parameter synthesis (e. g. EF-emptiness, EF-synthesis, AF-emptiness, AG-universality, etc). For example, given a PTA $\mathcal{A}$ and a subset $\mathcal{G}$ of its locations,

1. EF-emptiness asks: “is the set of parameter valuations v such that there exists a run of $\mathcal{A}[v]$ reaching a location $l \in \mathcal{G}$ empty?”

2. AF-emptiness asks: “is the set of parameter valuations v such that all runs of $\mathcal{A}[v]$ reach a location $l \in \mathcal{G}$ empty?”

3. EF-universality asks: “are all parameter valuations v such that there exists a run of $\mathcal{A}[v]$ reaching a location $l \in \mathcal{G}$?”

4. AF-universality asks: “are all parameter valuations v such that all runs of $\mathcal{A}[v]$ reach a location $l \in \mathcal{G}$?”

5. EF-synthesis does: “synthesize all parameter valuations v for which a run reaches a given location $l \in \mathcal{G}$.”

6. AF-synthesis does: “synthesize all parameter valuations v for which all runs reach a given location $l \in \mathcal{G}$.”

7. ED-emptiness asks: “is the set of parameters valuations v such that at least one run of $\mathcal{A}[v]$ is deadlocked, i. e. has no discrete successor (possibly after some delay), empty?”

8. AD-emptiness asks: “is the set of parameters valuations v such that all runs of $\mathcal{A}[v]$ are deadlocked, i. e. have no discrete successor (possibly after some delay), empty?”

9. EC-emptiness asks: “is the set of parameters valuations v such that there exists at least one run of $\mathcal{A}[v]$ with an infinite number of discrete transitions, empty?”

10. AC-emptiness asks: “is the set of parameters valuations v such that all runs of $\mathcal{A}[v]$ have an infinite number of discrete transitions, empty?”

11. Additionally, language and trace set preservation emptiness (a membership problem) asks: “does there exist another valuation $v' \neq v$ such that the untimed languages of $\mathcal{A}[v]$ and $\mathcal{A}[v']$ are the same?”

Note that EF-, AF-, EG-, AG-, ED- and EC-emptiness are equivalent to AG-, EG-, AF-, EF-, AC- and AD-universality, respectively.

In this thesis, the EF-problem will be used repeatedly in several chapters. Therefore, Section 2.9.5 will be devoted to the EF-problem. Other parameter synthesis problems will be mentioned inside other chapters. For additional properties please see e. g. [And15, AM15].

2.8.2 Decidability of PTA

In this section, we assume that the reader is familiar with the computation theory such as halting problem, decidability, reducibility, etc. However, the reader can also refer to Appendix A for a brief recall of the computation theory (e. g. decidability, reducibility, 2-counter machine [Min67], etc).

In [ALR16b], it showed that Turing-recognizable languages (type-0 in Chomsky's hierarchy or recursively enumerable languages) are also recognizable by PTAs with some restrictions on the number of clocks and parameters. Furthermore, a Turing-machine can be simulated by a 2-counter machine [Min67], which can be simulated by a PTA. Thanks to these simulations, we can prove a certain problem of PTA is undecidable by reducing the halting problem for 2-counter machines to this problem, which is known as undecidable.

The beginning of this section will be dedicated to the investigation of PTA's decidability. Following will show the attempts of simplifying and reducing its syntactic to have simpler subclasses with the hope of making them decidable.

Decidability of simple PTAs

Let us recall some decision problems for PTA. The EF-emptiness problem is undecidable [AHV93], even with a single real-valued parameter [Mil00], with only strict constraints [Doy07] and with a single integer-valued parameter [BBLS15]. The AF-emptiness problem is undecidable even with a single bounded parameter [JLR15] which is deduced from the proof of undecidability of L/U-PTA. Language and trace preservation emptiness problems are undecidable even with a single bounded parameter [AM15]. The EG and AG-emptiness problems are undecidable for PTAs, the AF-emptiness problem is undecidable for bounded PTAs, and the EF-universality problem is undecidable for bounded PTAs and for PTAs [ALR16a]. The EC-emptiness, ED-emptiness and EG-emptiness problems are undecidable for PTAs with 3 clocks and a bounded single parameter, with 3 clocks and 2 bounded parameters and with 4 clocks and 3 bounded parameters respectively [AL17]. (See more in e. g., [Mil00, Doy07, BO14, BBLS15, And15, AL17]).

Many attempts were dedicated to find the thin boundary between decidability and undecidability showing that a PTA will reach undecidability at three parametric clocks and only one rational-valued parameter, or only one parametric clock, three non-parametric clocks and one rational-valued parameter [Mil00], or three parametric clocks and one integer-valued parameter [BBLS15].

In general, bounding time, bounding the number of parameters or the domain of the parameters does not lead to any decidability. But by restricting the number of clocks, the use of clocks (compared or not with the parameters), and the use of parameters (e. g. used only as upper or lower bounds) can lead to the decidability of some problems. Research around PTAs since then consisted mainly in either exhibiting subclasses of PTAs for which interesting problems become decidable, or devising efficient semi-algorithms that would terminate "often enough" to be useful.

Decidability of subclasses of PTA

L/U-PTA A famous subclass of PTA is L/U PTA [HRSV02] where each parameter can be used only either as upper bounds or as lower bounds, and for which the EF-emptiness problem (see section 2.9.5) becomes decidable (additional results [BL09, JLR15, AL17]).

In [BL09], further problems were shown to be decidable for L/U PTA, including the emptiness and the universality problem for infinite runs properties ("do all parameter valuations have an infinite accepting run?"), for integer parameter valuations.

In [JLR15], however, it was shown that the solution to the EF-synthesis problem for L/U PTA cannot be represented as a finite union of polyhedra, hence strongly limiting the practical interest of L/U PTA. Orthogonal to syntactical restrictions on the model is the search for restrictions on the parameter domain: in [JLR15], an algorithm is proposed to synthesize integer parameter valuations in a bounded domain. This is of course decidable, and the authors devise two symbolic algorithms that perform better than enumeration.

In [ALR16a], EF-emptiness, EF-universality and AF-emptiness problems were again considered in solving several open problems for PTAs and their subclasses. Based on results proved for the new subclass called Integer-Points Parametric Timed Automata or (IP-PTAs) in this paper, and an original proof for the undecidability of the EF-emptiness problem for general PTAs with a single bounded rational-valued parameter and only non-strict constraints, the authors proved that the EF-emptiness and EF-universality problems are decidable, and the AF-emptiness problem is undecidable for bounded L/U-PTA.

In [AL17], the authors concentrate on liveness properties including EC-emptiness, ED-emptiness and EG-emptiness. Unfortunately, most of the problems were shown to be undecidable, except the EC-emptiness problem is proved to be decidable for L/U-PTA. The ED-emptiness and EG-emptiness problems are undecidable for bounded PTAs with 3 clocks and 2 parameters and with 4 clocks and 3 parameters respectively. Furthermore, in the paper, bounded L/U-PTAs are divided into two smaller subclasses called bounded open L/U-PTAs and bounded closed L/U-PTAs (depending on the bounded parameter domain interval). For closed bounded L/U-PTAs, The EC-emptiness and EG-emptiness problem are decidable and The ED-emptiness problem is undecidable with 3 clocks and 2 parameters. For open bounded L/U-PTAs, the ED-emptiness and EG-emptiness problems are undecidable with 3 clocks and 2 parameters and with 4 clocks and 4 parameters respectively. The EC-emptiness is still in open question.

Note that, bounded PTAs are a subclass of PTAs while bounded L/U-PTAs are incomparable of terms of expressiveness with L/U-PTAs [ALR16b], which means the undecidability results for bounded L/U-PTAs cannot be automatically extended to L/U-PTAs; conversely, decidability results for L/U-PTAs cannot be automatically extended to bounded L/U-PTAs.

L-PTA and U-PTA Lower-bound PTAs and upper-bound PTAs, abbreviated to L-PTAs and U-PTAs, are two subclasses defined in [BL09], where all parameters are always lower bounds, and upper bounds respectively. They are both very promising open classes with many undecidable problems for L/U-PTAs still remain open for them. The only problem is given in [AM15], which shows that the language-preservation problem is decidable for deterministic U-PTA

and deterministic L-PTA with a single integer-valued parameter and undecidable for L/U-PTA.

IP-PTA Integer-Points parametric Timed Automata [ALR16a] IP-PTAs for short, are a subclass of PTAs with bounded rational-value parameters. EF-emptiness, EF-universality and AF-emptiness are undecidable for both IP-PTAs and bounded IP-PTAs except the EF-emptiness problem which was proved to be decidable for bounded IP-PTAs. This is also known as the first syntactic restrictions making the reachability emptiness problem decidable for PTAs besides L/U-PTAs (see more in [ALR16a]).

Future subclasses Although PTAs suffer from many problems being undecidable in theory, there are many case studies using PTAs found to terminate in practice. The question is there are future subclasses of PTAs, for which the problems above become decidable.

2.9 Parameter synthesis

We recall in this section the algorithms and synthesis problems that are used repeatedly throughout the thesis.

2.9.1 The good parameters problem

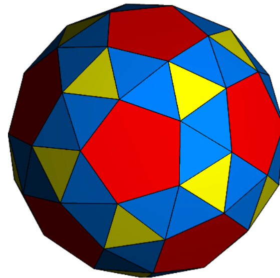


Figure 2.14 – An example of parameter domain in multi-dimensions (hyperrectangle or polyhedron)

The good parameters problem is a parameter synthesis problem that synthesizes *a set of values of the timing parameters* guaranteeing that the system behaves well and avoids any bad behavior [FJK08, AS13]. Note that, the set of values of the timing parameters is usually called as a hyperrectangle (resp. polyhedron [BY03]) parameter domain with each parameter corresponds with a dimension of the hyperrectangle (resp. polyhedron), and with n dimension we will have the polyhedron as Fig. 2.14 [1] above. For simplicity purposes, we only use

[1]The polyhedron picture in this section is obtained from : http://http://commons.wikimedia.org/wiki/File:Uniform_polyhedron-53-s012.png

rectangular parameter domains for our examples, such as the example in Fig. 2.15.

> **Good parameters problem:**
> PROBLEM: Given a concurrent real time system and a hyperrectangle parameter domain $D_0 \subseteq R_+^M$, what is the largest set of parameter values within D_0 for which the system is safe?

2.9.2 The Inverse problem

The inverse problem [AS13] is a subproblem of the good parameters problem above Section 2.9.1.

> **Inverse problem:**
> PROBLEM: Given a parametric timed automaton $\mathcal{A}$ and a reference valuation v_0, find a constraint K_0 on the parameters such that:
>
> - $v_0 \models K_0$, and
> - for all $v \models K_0$ the trace sets of $\mathcal{A}[v_0]$ and $\mathcal{A}[v]$ are the same.

This problem considers an equality of trace sets between $\mathcal{A}[v_0]$ and $\mathcal{A}[v]$, and thus guarantees a "time-abstract" equivalence between the behavior of $\mathcal{A}[v_0]$ and $\mathcal{A}[v]$.

The synthesizing procedure starts with a given valuation v_0 of the system, then it synthesizes a constraint K_0 with the same trace set as the one of v_0. The benefit of that is to give a robustness criterion by guaranteeing the same behavior around v_0.

2.9.3 The Inverse Method

The *inverse method* (IM) [AS13] generalizes the behavior of $\mathcal{A}[v]$ in the form of a *tile*, i. e. a parameter constraint K where the discrete behavior is uniform (see Fig. 2.15, where $K = \mathsf{IM}(\mathcal{A}, v)$). That is, for any point v' satisfying K, the trace sets of $\mathcal{A}[v']$ and $\mathcal{A}[v]$ are equal. Hence any linear-time property (expressed in, e. g. LTL) valid in $\mathcal{A}[v]$ is also valid in $\mathcal{A}[v']$.

An application of IM is to derive robustness conditions for the system. The study of the robustness in real-time systems (see, e. g. [Mar11]) aims at deciding whether infinitesimal variations of time (due to, e. g. slightly extended or shrunk deadlines, or clock drifts) may impact the overall, discrete system behavior. However, IM may not terminate [ACEF09]. Indeed, parameter synthesis for PTA is known to be undecidable [AF10]. Nevertheless, it behaves "well" in the sense that it terminates for all case studies considered in Chapter 3, except for small examples designed on purpose to show non-termination.

Note that, in general, tiles have no predefined "shape": they are general polyhedra in $|P|$ dimensions that can have arbitrary size, number of vertices, and edge slope. The computation time of IM also greatly varies, from milliseconds

to several hours, depending on the complexity of the model, and the size of the trace set.

2.9.4 The Behavioral Cartography

Given a PTA $\mathcal{A}$ and a bounded parameter domain D (usually a hyperrectangle in $|P|$ dimensions), the *behavioral cartography* (BC) [AF10] repeatedly calls IM on (some of the) integer or rational-valued points of D (of which there is a finite number), so as to cover D with tiles. The result gives a tiling of D such that the discrete behavior (trace set) is uniform in each tile.

In Fig. 2.15, BC first considers point v, and computes $K = \mathsf{IM}(\mathcal{A}, v)$. Then, BC iterates on the subsequent points, all already covered by K, until it meets v'', that is not yet covered. Hence, BC will then compute $\mathsf{IM}(\mathcal{A}, v'')$, and so on, until all points in D are covered.

BC can be used for several applications: first, it identifies the system robustness in the sense that, in each tile, parameters can vary as long as they remain in the tile, without impacting the system's discrete behavior. Second, BC can be used to perform parameter optimization; the weakest conditions of the input signal of an industrial asynchronous memory circuit (SPSMALL) were derived using BC [AS13]. Third, given a set of linear time properties (i. e. that can be verified on the trace set), it suffices to compute only once BC, and then to check each property on the trace set generated for each tile in order to know a complete (or nearly complete) set of parameter valuations satisfying each property.

Remark. BC does not guarantee the full, dense coverage of D for two reasons:

1. IM may not terminate, as the corresponding problem is undecidable [AM15]. In practice, in our implementation of BC in Chapter 3, this is addressed using a timeout: if $\mathsf{IM}(\mathcal{A}, v)$ does not terminate within some time bound, BC switches to the next point, and v will (most probably) never be covered. However, although it was shown possible in theory, this never happened in any of our experiments.

2. IM generalizes integer points in the form of dense, rational-valued constraints, but it could happen in rare cases that some tiles do not contain any integer points. This sometimes happened in our experiments (e. g. the

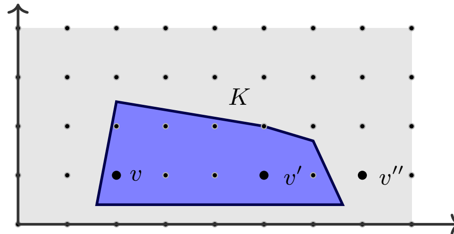
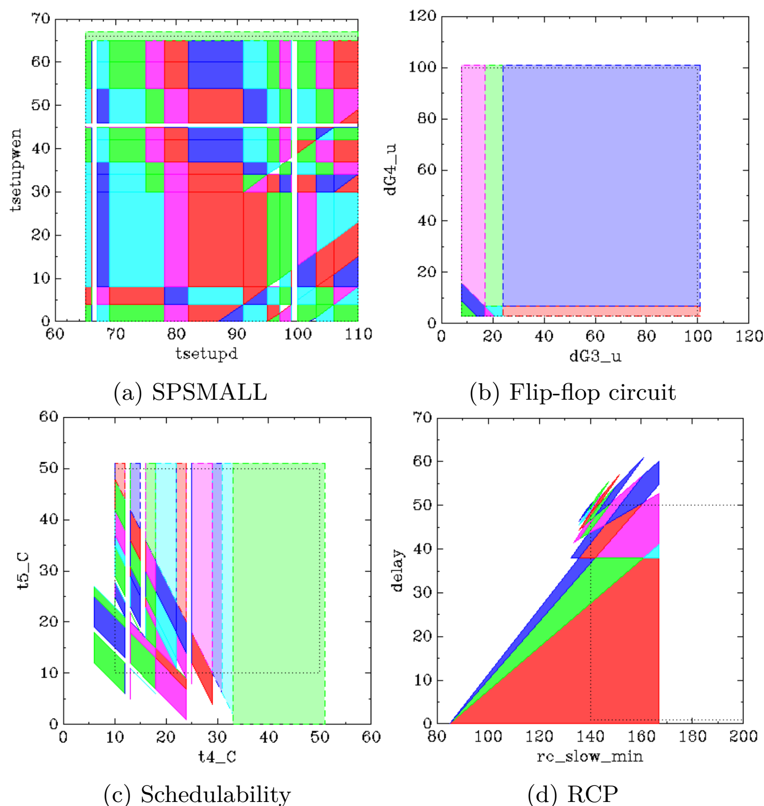


Figure 2.15 – Graphical example

small white zones in BC in Fig. 2.16a, around $x = 100$ and $y = 55$); usually, calling BC on multiples of $\frac{1}{3}$ instead of integers was empirically shown to be sufficient in most cases (although in theory there might be an infinite number of tiles in a bounded domain). Conversely, note that BC frequently covers (parts of) the parametric space beyond D which is a predefined dotted rectangle in Fig. 2.16; for instance, the case in Figs. 2.16b to 2.16d (in Fig. 2.16b, the entire parametric space is even covered).



(a) SPSMALL

(b) Flip-flop circuit

(c) Schedulability

(d) RCP

Figure 2.16 – Examples of graphical behavioral cartographies in 2 dimensions

Also note that the motivation for considering integer points is that, in most cases, considering integers is sufficient to cover entirely (or almost entirely) the domain D. However, as said above, our implementation allows any "step" instead of integers (e. g. multiples of $\frac{1}{3}$).

2.9.5 EF-problems

We recall below two classical problems, as formalized in [JLR15].

EF-emptiness problem:
INPUT: A PTA $\mathcal{A}$ and a location l_{bad}
PROBLEM: Is the set of parameter valuations v such that $\mathcal{A}[v]$ reaches l_{bad} empty?

EF-synthesis problem:
INPUT: A PTA $\mathcal{A}$ and a location l_{bad}
PROBLEM: Compute the set of parameter valuations v such that $\mathcal{A}[v]$ reaches l_{bad}.

The EF-emptiness problem is undecidable [AHV93], and the set of parameter valuations solving the EF-synthesis problem cannot be computed in general. In [JLR15], the following semi-algorithm is proposed, that gives a complete answer to EF-synthesis when it terminates.

Throughout the thesis, we call "EFsynth" the parameter synthesis semi-algorithm that solves the EF-synthesis problem. We will be interested in the reachability of *bad locations*. Given a PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, K_0, I, E)$, we assume a special location $l_{bad} \in L$; without loss of generality, we assume that this location is unique (the case with several bad locations can be reduced to one only using additional transitions to l_{bad}).

Let us recall the semi-algorithms in Fig. 2.17 for the EFsynth problem in [JLR15], where S represents a passed list of symbolic states. S begins with an empty list and then records the symbolic states that have already been explored on a given path:

$$\mathsf{EFsynth}_{l_{bad}}((l,C),S) = \begin{cases} C{\downarrow_P} & \text{if } l = l_{bad} \\ \emptyset & \text{if } (l,C) \in S \\ \bigcup_{s' \in \mathsf{Succ}((l,C))} \mathsf{EFsynth}_{l_{bad}}\big(s', S \cup \{(l,C)\}\big) & \text{otherwise} \end{cases}$$

Figure 2.17 – EFsynth algorithm

Example 15. Consider the PTA $\mathcal{A}_1$ in Fig. 2.18 [JLR15], with clocks x and y and parameters a and b. Then $\mathsf{EFsynth}_{l_2}(s_0, \emptyset)$ does not terminate, and neither does it if the range of the parameters is bounded from above (e. g. $a, b \in [0, 50]$).

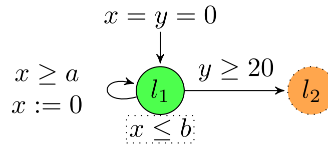


Figure 2.18 – An example of a PTA $\mathcal{A}_1$ [JLR15]

From the proof of correctness of EFsynth in [JLR15], one can infer that the result of EFsynth is still a (possibly incomplete) answer to the EF-synthesis problem even when the algorithm is artificially stopped before its termination. By

artificially stopping EFsynth, we mean bounding the recursion depth: when the depth indeed exceeds some bound, we replace the recursive call $\mathsf{EFsynth}_{l_{bad}}(s', S \cup \{(l, C)\})$ with $\bot$.

Proposition 2.9.1 ([JLR15]). *Let K be the result of $\mathsf{EFsynth}_{l_{bad}}(s_0, \emptyset)$ when EFsynth is stopped after being recursively called a bounded number of times. For all $v \models K$, l_{bad} is reachable in $\mathcal{A}[v]$.*

Then, BC can give a (possibly incomplete) solution to the EF-synthesis problem, by returning the union of all constraints for which the desired location is reachable.

2.10 IMITATOR

IMITATOR[2] [AFKS12, And09] stands for Inverse Method for Inferring Time AbstracT behaviOR, and implements of the InverseMethod algorithm described in [ACEF09], and also a tool for efficient synthesis for Timed Automata. Technically, it is based on the inverse method [AFKS12] and the behavioral cartography for PTA [AF10]. IMITATOR takes as input a network of PTAs and e. g. a reference valuation v_0, and then it synthesizes a constraint K solving the inverse problem. In particular, it generalizes a concrete behavior of a PTA, by synthesizing constraints on the parameters.

Figure 2.19 – Functional view of IMITATOR

This tool has been originally developed by Étienne André, in collaboration with various contributors.

Ocaml: A programming language is developed at INRIA in France [Har05, MMH13]. Since its first release in 1996, it has excelled in a variety of application domains and supports functional, imperative and object-oriented styles of programming which ease the development of flexible and reliable software. Thus Ocaml has become one of the most popular functional programming languages in the world, which has many powerful features such as functional programming, pattern matching, type inference, garbage collection, object-oriented programming, safety, strongly and statically typed, type inference, polymorphism, modules programs, separate compilation, etc. It is clear that OCaml is a good choice for systems where need resource usage, predictability, and performance. Until now, Ocaml language improvements have been frequently added to support the growing commercial and academic purposes.

[2] https://www.imitator.fr/

2.11 Efficient verification

Since the early stages, one of the main drawbacks of model checking has been the state explosion problem. Therefore, much of research in model checking over the past few decades has involved developing techniques for dealing with this problem such as *symbolic model checking (see Section 2.5.4), abstraction verification, compositional verification, symmetry verification, on-the-fly technique, partial order reduction, bounded model checking, SAT bounded model checking*, etc.

Besides symbolic model checking and on-the-fly techniques that are used in all our algorithms. We also use abstraction verification in modeling our systems and compositional verification for our distributed schemes. Hence, we give below a short introduction to these techniques.

2.11.1 On-the-fly verification

This approach avoids constructing the entire state space of the model in the memory and the states are generated only when needed. Thus it decreases the computation time and allows us to analyze larger models [FJJV96, CVWY92].

2.11.2 Abstracted verification

With abstraction [Lon93], we try to simplify our models by hiding details, replace complex data types with simpler abstract ones, or simplify some of the timing behavior of the components. Indeed, verifying the simplified models is faster than verifying the original ones. Note that we must establish a relationship between the abstract models and the original ones, so that correctness at the abstract level will imply correctness for the original system. Furthermore, there must be a balance between hiding information and the need to be able to prove the specification.

2.11.3 Compositional verification

In compositional verification [Lon93], a given system model is split up into sub-models in order to faster verify. In fact, the key idea is to verify the properties of individual components of the system but still guarantee the properties hold in the whole system, and then use them to prove the overall system specification. Note that it requires knowledge of how components of the design contribute to satisfying the given specification.

2.12 Parallel computing

Although many approaches have been proposed in Section 2.11 to make model checking more efficient, the state explosion problem still occupies the computer hardware resources (both processor and memory) at most linearly. Then using a model checker based on parallel (shared memory) and distributed (message passing) computing will be more efficient to handle large state spaces. This section

will be dedicated to the basis of parallel computation, especially distributed computing and message passing interface which are used in this thesis.

Parallel computing [**B+10**] is a form of computation in which many calculations are carried out simultaneously by parallel systems.

A parallel system is a system having more than one processor with parallel processing abilities. This system is based on the connections between processors and memories, and SIMD and MIMD are two main kinds of the parallel system which are from four classifications defined by Flynn:

- **Single Instruction Single Data Stream (SISD)**: Traditional sequential architecture. At one time, one instruction operates on one data.
- **Single Instruction Multiple Data Stream (SIMD)**: Multiple data streams are executed by only one instruction.
- **Multiple Instruction, Single Data stream (MISD)**: Multiple instructions operate on a single data stream. This architecture is often used for fault tolerance.
- **Multiple Instruction Multiple Data Stream (MIMD)**: Multiple processors execute different instructions on different data streams, which leads to the flexibility in parallel processing. *Parallel and distributed systems* are also recognized as MIMD architectures.

A parallel programming model can be executed on a parallel system. It includes design and coding applications. The purpose is to parallelize a sequential program in order to gain the ability to solve large complex problems or reduce execution time or both. Moreover, the main idea behind parallel programming is to parallelize the program to distribute work across multiple processors, and finally gather all results from these processors. There are several parallel programming models in common use, for instance:

- **Shared memory**: Multiple processors can communicate together by reading and writing from a shared memory. It means data in shared memory can be accessed by all the processors.
- **Distributed memory**: Unlike shared memory systems, each processor has access to its own memory space.

Knowing them clearly could help to build the right algorithm. In the thesis, we use distributed memory model for our algorithms, and processes communicate with each other by using message passing.

Message Passing Interface - MPI [Bar, GFB+04] stands for Message Passing Interface, is a standardized and portable message-passing system designed by a group of researchers from academia and industry to supply programmers programming on a variety of parallel platforms.

The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for programming parallel applications. MPI provides language bindings for C, C++, Fortran-77, and Fortran-90, non-official bindings exist for python and Ocaml. There exist several open-source implementations but the most widely used are being Open-MPI and MPICH2.

IMITATOR is written in Ocaml; Hence, we used "OcamlMPI" which is an implementation of Open-MPI for this project.

It is necessary to get familiar with basic MPI definitions which are used in next chapters:

Note that, MPI uses objects called communicators and groups to define which collection of processes may communicate with each other

- **Rank**: It is a unique integer identifier for a process assigned by the system when the process initializes and used by the programmer to specify the source and destination of messages. Ranks are contiguous and begin at zero.
- **Communicator**: This defines the group which may communicate, possesses its own unique identifier. Most MPI subroutines require to specify the communicator as an argument.
- **Group**: It is an ordered set of processes and has its own unique identifier, associated with a communicator.

In the next chapters, we will detail the main contributions of the thesis. The succinct Section 2.9 and Section 2.8 should be sufficient for the reader to understand all ideas and concepts that the thesis conveys.

Chapter 3

Distributed verification of parametric real-time systems

Contents

3.1 Introduction **53**
3.2 Static domain decomposition **55**
3.3 Master-worker point distribution algorithms **57**
3.3.1 Principle: master-worker 57
3.3.2 An abstract algorithm for the master 58
3.3.3 Sequential point distribution 59
3.3.4 Random + sequential point distribution 60
3.3.5 Shuffle point distribution 61
3.4 Dynamic domain decomposition **62**
3.4.1 Master algorithm 63
3.4.2 Worker algorithm 64
3.4.3 An additional heuristic 65
3.5 Experiments **66**
3.6 Conclusion **72**

3.1 Introduction

Parametric verification is costly, so we try to distribute it in order to make it faster. The behavioral cartography (BC) of PTA was presented in Section 2.9.4

and it seems to be the easiest algorithm to extend in a distributed manner. Hence, this chapter is devoted to distributing BC on a cluster. Then we will address EF in Chapter 4.

In [ACE14], the authors sketched two master-worker point distribution algorithms to compute BC in a distributed fashion. Here, we present enhanced distributed algorithms to compute the cartography efficiently. Experimental results show that our new algorithms (implemented in the IMITATOR tool [AFKS12]) significantly outperform previous distribution techniques.

Contribution The goal of this chapter is to propose efficient distributed algorithms to compute BC efficiently using parallel, distributed computing resources. Our contributions are as follows:

1. We formalize the existing point-by-point distribution algorithms (Sequential and Random), that were only informally sketched in [ACE14].

2. Then, our main contribution is to propose three new distributed algorithms to speed up the cartography: the first one (Static) is a static domain decomposition scheme, where each node works independently on its own parameter subdomain; the second one (Shuffle) addresses the drawbacks of Sequential and Random; finally, the third one (Subdomain) is a new master-worker, dynamic, distributed domain decomposition process.

3. We then evaluate our algorithms on real-time case studies. In all cases, our new algorithms Shuffle and (a variant of) Subdomain outperform the algorithms of [ACE14]. We also discuss how to choose the appropriate algorithm depending on the case study.

Related works The design of efficient parameter synthesis techniques has been tackled in various works, e. g. using SMT-based model checking techniques [CGMT13], or using symbolic techniques for integer synthesis [JLR15]. BC helps to quantify the system robustness; this has also been tackled using the "ASAP" semantics [DWDR05] (see, e. g. [Mar11] for a survey), but usually in only one dimension (a single variation δ of the timing delays is considered, whereas BC allows as many dimensions as parameters). To the best of our knowledge, with the exception of [ACE14], distributed computing techniques were not applied yet to parameter synthesis for PTA.

Formal verification can be made in parallel in two ways: modeling languages can be designed to be easy to use in a distributed fashion, or the verification algorithms themselves can be parallelized. Our approach sits in the second category. In recent years, some model checkers were extended to parallel computing, i. e. running on multicore computers. This is the case of PKind [KT11], APMC (a probabilistic model checker) [HBE+10], and FDR3 (for CSP refinement checking). More recently, two algorithms were proposed to address multi-core LTL verification [ELPvdP12] and emptiness checking of timed Büchi automata [LOD+13]. However, with the exception of FDR3 (that can run either on multicore or on

clusters), these works run verification on multicore computers (with a shared memory) whereas our primary goal is to run verification on a cluster in order to cope with memory consumption problem (where each node has its own memory). Furthermore, there are other works with shared memory such as the novel distributed memory algorithm for SMT-based parameter synthesis in [BBD+16], or swarm verification for time optimal reachability (TOR) in UPPAAL [ZNL16a, ZNL16b].

Outline We briefly define in Section 3.2 the static domain decomposition algorithm (Static). Then, we formalize in Section 3.3 the master-worker scheme and the two point distribution algorithms of [ACE14]; we also introduce a third point distribution algorithm (Shuffle). We introduce in Section 3.4 our new dynamic domain decomposition algorithm (Subdomain). We conduct experiments in Section 3.5 and conclude in Section 3.6.

3.2 Static domain decomposition

In order to tackle larger case studies, our objective is to take advantage of the iterative nature of the cartography (in contrast to most, if not all, other known parameter synthesis algorithms), and to distribute it on N processes. There is no theoretical obstacle in doing so, since all calls to IM are independent from each other. The challenge is rather to select efficiently the points on which IM is called, so that as few redundant constraints as possible are computed.

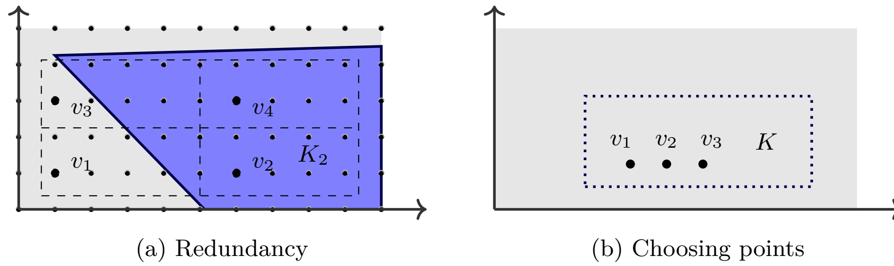
In this section, we briefly describe a static domain decomposition ("Static"). That is, the rectangle D is split into N subdomains, and then each process is responsible for handling its own subdomain in an independent manner (with no communication). This domain decomposition method is often used for regular data distributions, where all subdomains require the same processing time, and preferably on domain shapes such as rectangles or hypercubes, that can easily be mapped on a grid of processes.

Each node i performs the following procedure:

1. split D into N subdomains. Alternatively, a single node could perform the split and then send to each other node its own subdomain (at the cost of additional communications).
2. execute BC on the ith subdomain, i. e. iteratively select integer points and call IM until all integer points in the ith subdomain are covered by tiles.

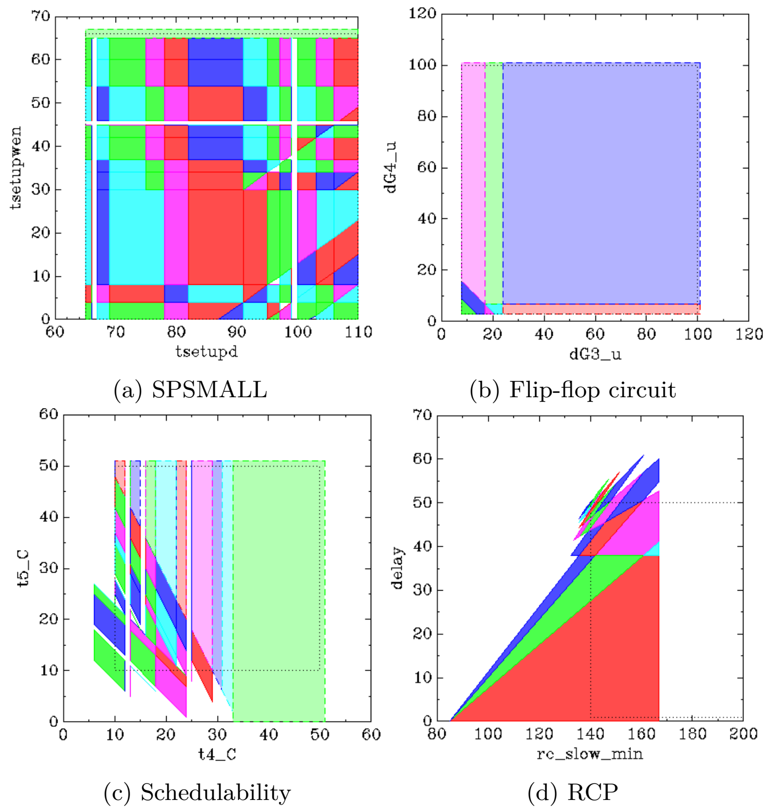
For example, in Fig. 3.1a, the domain D (the external dashed rectangle) is split into four equal subdomains (the four internal dashed rectangles); v_i, $1 \leq i \leq 4$ represents a possible first point on which to call IM in each subdomain. (K_2 in Fig. 3.1a and Fig. 3.1b will be used later on.)

This static decomposition is straightforward but is not satisfactory for BC for three main reasons.



(a) Redundancy (b) Choosing points

Figure 3.1 – Graphical representations and challenges



(a) SPSMALL (b) Flip-flop circuit

(c) Schedulability (d) RCP

Figure 3.2 – Examples of graphical behavioral cartographies in 2 dimensions

First, the general "shape" of the cartography is entirely arbitrary and unknown beforehand, since tiles can themselves have any shape. Fig. 3.2 gives examples of cartographies in 2 parameter dimensions: although the geometrical distribution of the tiles of Fig. 3.2a within D is rather homogeneous, this is not true at all for the others. For example, splitting the domain of Fig. 3.2b (resp. Fig. 3.2d) into four equal parts would be very unfair for the node responsible of

the lower-left (resp. upper-right) subdomain, since most tiles are concentrated there; this would also be inefficient, since the other nodes will rapidly become idle.

Second, the geometrical distribution of the tiles says nothing on the *time* necessary to compute each tile. Recall that the computation of IM can be very long (up to several hours). Even when the tiles are homogeneously located within D, some tiles may require much more time than others. For example, in Fig. 3.2a (where the geometrical distribution of the tiles is rather homogeneous), it could happen that the bottom-left tiles require much more time than others, resulting in this node to work much longer, while the other nodes would rapidly finish their duty. Again, this would result in a loss of efficiency due to load unbalance since not all of the nodes are working actively.

Third, the absence of communication between nodes may result in redundant computations. Let us go back to the example of cartography in Fig. 3.1a. Assume that node 2 finishes first the computation of a tile, say K_2. This tile not only covers the entire subdomain of node 2, leading to the termination of process 2, but it also covers node 4's subdomain entirely and a large part of node 2's subdomain. Without communication, these nodes will keep working without knowing that their subdomain has already been covered. In contrast, a smarter distribution scheme should be such that, in this situation, nodes 2, 3 and 4 would go to help node 1 finish its (not much covered yet) subdomain. We will address this efficiency issue in the remainder of this chapter.

3.3 Master-worker point distribution algorithms

We formalize the abstract algorithm for the master (Section 3.3.2), the Sequential (Section 3.3.3) and the Random point distribution (Section 3.3.4). Additionally, we introduce a new point distribution Shuffle (Section 3.3.5).

3.3.1 Principle: master-worker

Workers ask the master for a point v, then execute $\mathsf{IM}(\mathcal{A}, v)$, and finally send the corresponding result K to the master. The master does not call IM itself, but instead distributes points to the workers. Whereas this may be a loss of efficiency when using few processes, this shall be compensated for a large number of processes. Moreover, this parallel computation scheme balances the load between workers automatically.

Master tag	Argument	Worker tag	Argument
POINT(v)	parameter valuation	COMPLETED	-
STOP	-	NOTIFYPOINT(v)	parameter valuation
SUBDOMAIN(sd)	new subdomain	REQTILES	-
TILES(T)	latest tiles	RESULT(K)	constraint computed

Table 3.1 – Tags for master-worker communications

The master and workers communicate with each other by sending messages that are labeled using *tags*, using two asynchronous functions $\textit{send}(n, \textit{msg})$ and $\textit{receive}()$. Function $\textit{send}(n, \textit{msg})$ sends a tagged message *msg* to node n. Function $\textit{receive}()$ is a blocking function that waits until a message is received, and returns a pair $(n, \textit{msg})$, where *msg* is the tagged message that has been received from node n. Based on the tag of the message, receiving processes can decide what to do with the message itself. Note that workers never communicate with each other. We assume that messages are made of a tag and zero or one argument: for example, POINT(v) sends a POINT tag together with the parameter valuation v. We give the list of tags used throughout this chapter in Table 3.1.

3.3.2 An abstract algorithm for the master

We first formalize in Algorithm 2 the "abstract" master algorithm sketched in [ACE14]; this algorithm contains variation points that can be instantiated to give birth to concrete master algorithms. In this section, we only use the worker tag RESULT and the master tags POINT and STOP. The workers, formalized in Algorithm 1, only call the inverse method on the point they receive from the master, and send the result back, until a STOP tag is received.

Algorithm 1: Worker algorithm

```
  input    : PTA 𝒜
1 while true do
2 │ switch receive() do
3 │ │ case m, STOP: do Terminate
4 │ │ case m, POINT(v): do
5 │ │ │ K ← IM(𝒜, v)
6 │ │ │ send(m, RESULT(K))
```

Algorithm 2 takes as input a PTA $\mathcal{A}$, a parameter domain D and the number of processes N; it is also parameterized by a *point distribution mode* M. Each mode is responsible for instantiating the variation points to give birth to a concrete algorithm. The master starts by creating an empty set of tiles and then calls the mode initialization function $\mathsf{M}.\textit{initialize}()$, that initializes the various variables needed by the concrete algorithms (line 1). Then, the master sends a point to each node n; the way these points are chosen among D ($\mathsf{M}.\textit{choosePoint}()$) is decided by the mode (line 2). Then the master enters the main loop (line 3 to line 5): while there are uncovered points, every time a node n sends a constraint K and asks for work, the master stores the result in its list of tiles T; then, it selects a point according to M and sends it to n. Finally, once all integer points are covered, the master receives the last result from every node and sends STOP tags (line 6–line 7).

The way points are picked by the master to be distributed to the workers is a highly critical question. Choosing points in a wrong manner can lead to

Algorithm 2: Abstract algorithm for the master

input : PTA $\mathcal{A}$, domain D, number of processes N, mode M
output : Set of tiles T

```
// Initialization phase
```
1 $T \leftarrow \emptyset$; $\mathsf{M}.\mathit{initialize}()$
2 **foreach** *process* $n \in \{1, \ldots, N\}$ **do** $\mathit{send}(n, \text{POINT}(\mathsf{M}.\mathit{choosePoint}()))$
```
// Main phase
```
3 **while** *there are uncovered integer points in* D **do**
4 $\quad n, \text{RESULT}(K) \leftarrow \mathit{receive}()$; $T \leftarrow T \cup \{K\}$
5 $\quad \mathit{send}(n, \text{POINT}(\mathsf{M}.\mathit{choosePoint}()))$
```
// Finalization phase
```
6 **foreach** *process* $n \in \{1, \ldots, N\}$ **do**
7 $\quad n, \text{RESULT}(K) \leftarrow \mathit{receive}()$; $T \leftarrow T \cup \{K\}$; $\mathit{send}(n, \text{STOP})$
8 **return** T

a dramatic loss of efficiency. For example, choosing points very close to each other would most probably lead to the (redundant) computation of the same tile. This situation is depicted graphically in Fig. 3.1b, where points v_1, v_2, v_3 may yield the same tile K. In the next three subsections, we formalize three master modes; these modes will define additional global variables and must instantiate *initialize*() and *choosePoint*().

3.3.3 Sequential point distribution

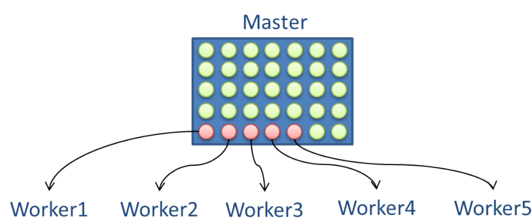


Figure 3.3 – Sequential algorithm illustration

The first point distribution algorithm (Sequential) is a direct extension of the monolithic (i. e. non-distributed) algorithm: as in the non-distributed BC, it enumerates all the points of D in a sequential manner starting from $\mathbf{0}$ as described in Fig. 3.3. Sequential assumes a function *nextPoint* that, given a parameter valuation v and a parameter domain D, returns the next point in D for some lexicographic order on the points of D. Sequential maintains a single global variable v_{prev}, storing the latest point sent to a worker. The initialization function Sequential.*initialize*() sets v_{prev} to a special value $\perp$ such that $\mathit{nextPoint}(\perp)$

returns the smallest point in D (e. g. $\mathbf{0}$ if $\mathbf{0} \in D$). Sequential.*choosePoint*() (given in Algorithm 3) returns the next point of D not covered by any tile yet.

Algorithm 3: Sequential.*choosePoint*()

variables : Point v_{prev} (global)
output : Point v

1. $v \leftarrow v_{prev}$
2. **repeat** $v \leftarrow nextPoint(v, D)$ **until** *v is not covered by any tile in T*
3. $v_{prev} \leftarrow v$; **return** v

The main advantage of Sequential is that it is inexpensive on the master's side.

Its main drawback is the risk of redundant computations by the workers, due to the situation depicted graphically in Fig. 3.1b: for instance, at the beginning, the N processes will ask for work, and the master will give them the first sequential N points, all very close to each other, with a high risk of redundant computation.

3.3.4 Random + sequential point distribution

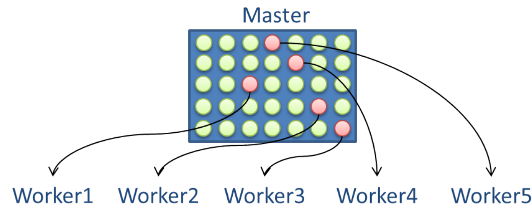


Figure 3.4 – Random algorithm illustration

The second point distribution algorithm (Random) selects points randomly as described in Fig. 3.4, and then in a second phase performs a sequential enumeration to check the full coverage of integers in D. This second phase is necessary to guarantee that all the integer points have been covered. The second phase starts after a given number *MAX* of consecutive failed attempts to find an uncovered point randomly. Indeed, simply stopping BC after *MAX* tries could give a probabilistic coverage (e. g. 99 %) of integer points, but cannot guarantee the full coverage. Since finding the points not covered by a list of tiles has no efficient practical solution, this sequential check is the only concrete option we have.

Random maintains two global variables. First, *seqPhase* acts as a flag to remember whether the algorithm is in the first or second phase. Second, v_{prev} stores the latest point sent to a worker (just as in Sequential). Random.*initialize*() initially sets *seqPhase* to `false` and v_{prev} to $\perp$.

We give Random.$choosePoint()$ in Algorithm 4. In the first phase (line 1 to line 7), Random.$choosePoint()$ randomly computes a point, and then checks whether it is covered by any tile; if not, it is returned. Otherwise, a second try is made, and so on, until the maximum number MAX of attempts is reached. In that latter case, it switches to the second phase (line 8 to line 11), consisting in a sequential enumeration of all the points just as in Sequential.$choosePoint()$.

Algorithm 4: Random.$choosePoint()$

```
   variables : Point v_prev, flag seqPhase, Int nbTries (global)
   output    : Point v
   // First phase
 1 if ¬seqPhase then
 2 |   nbTries ← 0
 3 |   while nbTries < MAX do
 4 |   |   v ← randomPoint(D)
 5 |   |   if v is not covered by any tile in T then  return v
 6 |   |   nbTries ← nbTries + 1
 7 |   seqPhase ← true
   // Second phase
 8 if seqPhase  then
 9 |   v ← v_prev
10 |   repeat v ← nextPoint(v)  until v is not covered by any tile in T
11 |   v_prev ← v  ;  return v
```

3.3.5 Shuffle point distribution

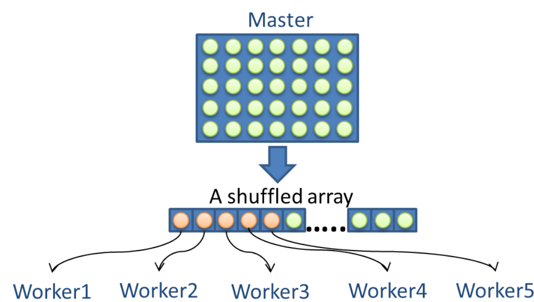


Figure 3.5 – Shuffle algorithm illustration

The main problem of Random is the fact that the second phase, necessary to check the full coverage of integers, may be costly and even useless if almost all the points have already been covered. To alleviate this problem, we propose a

new algorithm Shuffle that first computes statically a list of all integer points in D, then shuffles this list, and then selects the points of the shuffled list in a sequential manner. The sequential phase of Random is then dropped, at the cost of being able to compute, store statically and shuffle a large quantity of points.

Shuffle maintains a single global variable, i. e. the list *allPoints* of all the points in D that has been shuffled as described in Fig. 3.5. The Shuffle.*initialize*() function assigns *shuffle*(*allIntegers*(D)) to *allPoints*. (We assume here that function *allIntegers*(D) returns the list of all the integer points of D, and function *shuffle*(L) shuffles the elements of a list L.)

Then, the Shuffle.*choosePoint*() function simply consists in selecting the next uncovered point in *allPoints*. That is, it performs *pop*(*allPoints*), until the point output is not covered by any tile, in which case it returns it (we assume here that function *pop*(L) pops the first element of the list L and returns it).

3.4 Dynamic domain decomposition

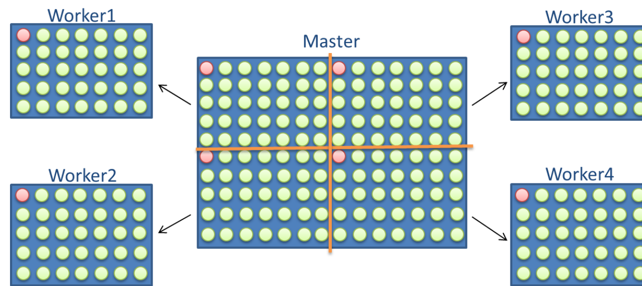


Figure 3.6 – Subdomain algorithm illustration

The most intuitive solution for distributing BC is the Static distribution scheme of Section 3.2, i. e. to split D into N subdomains, and then ask each process to handle its own subdomain in an independent manner as described in Fig. 3.6. As said in Section 3.2, this may lead to inefficient computations (which will be confirmed by our experiments in Section 3.5). Still, we use this idea to set up a *dynamic* domain decomposition algorithm. This algorithm is different from the previous ones, in the sense that it does not fit in the abstract master algorithm formalized in Section 3.3.2.

Initially, the master splits in D into N subdomains, and distributes the subdomains to the workers. In contrast to the algorithms of Section 3.3, the workers are now responsible for checking whether all the points in their subdomain have been covered yet or not. This mechanism reduces the load on the master without leading to redundant point coverage checks. Then, when a worker has covered all the integer points in its subdomain (because the points are covered by tiles computed either by this worker, or by other workers), it informs the master; the master dynamically splits a subdomain (typically, one that has only

been covered a little) and sends it back to the idle worker.

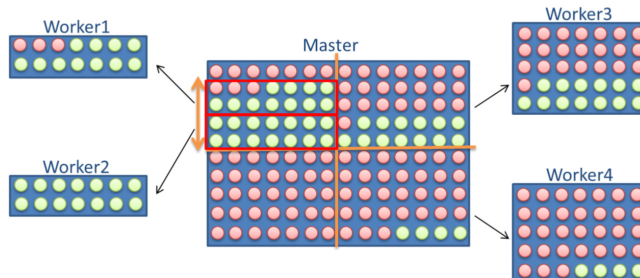


Figure 3.7 – Subdomain algorithm with splitting process illustration

The main idea is that the master is responsible for handling the dynamic distribution of the subdomains (including detecting the slowest workers to split their subdomain as described in Fig. 3.7), whereas the workers are responsible for covering all the points in their subdomain in a sequential manner. There is no need for more complex algorithms, since each worker is working on its own in its own subdomain.

3.4.1 Master algorithm

In the following, we assume several functions. We believe that understanding the role of these functions is straightforward; in practice, they lead to very tricky implementation issues (especially for the *split* function with arbitrary numbers of processes and parameter dimensions).

We give the master algorithm in Algorithm 5. Besides the list of tiles T, the master maintains two arrays of size N: the array *SD* associating with each node its current subdomain, and the array *currentPoints* associating with each node its latest known point (used to understand how advanced a worker is in its subdomain). These two arrays are initialized using the function *initialSplit* that splits D into N subdomains (line 1). Then the master sends its subdomain (line 2) to each node.

The algorithm then enters its main phase (line 3 to line 10). The master waits for incoming messages received via the asynchronous, blocking function *receive*(). If a new point is received (line 5), the master updates the *currentPoints* array (this is needed to perform splits using the most up-to-date data). If a result is received (line 6), the master stores it. If a request for tiles is received (line 7), the master sends all the tiles back so that n can update its local list. For efficiency purposes, in our implementation, the master only sends the new tiles since n's latest request (which is ensured using additional queue data structures). The local list is necessary to detect whether a point in the worker's subdomain is covered by a tile computed by another worker. If the master is notified that a worker n has completed its subdomain, i. e. all of its points have been covered (line 8), the master finds out which subdomain is the least covered,

Algorithm 5: Subdomain: Master

```
   input    : PTA 𝒜, domain D, number of processes N
   output   : Set of tiles T
   // Initialization phase
1  T ← ∅ ;  SD, currentPoints ← initialSplit(D, N)
2  foreach process n ∈ {1, ..., N} do  send(n, SUBDOMAIN(SD[n]))
   // Main phase
3  while a subdomain in SD can be split do
4      switch receive() do
5          case n, NOTIFYPOINT(v): do  currentPoints[n] ← v
6          case n, RESULT(K): do  T ← T ∪ {K}
7          case n, REQTILES: do  send(n, TILES(T))
8          case n, COMPLETED: do
9              n', sd_1, sd_2 ← split(SD, currentPoints, n)
10             send(n, SUBDOMAIN(sd_1)) ; send(n', SUBDOMAIN(sd_2))
   // Finalization phase
11 finished ← 0;
12 while finished < n do
13     switch receive() do
14         case n, RESULT(K): do  T ← T ∪ {K}
15         case n, COMPLETED: do  send(n, STOP) ; finished++
16         case *: do  do nothing
17 return T
```

i. e. which workers are the most in need for assistance. This is performed by *split*(SD, *currentPoints*, n), that returns the node n' needing help, and two new subdomains sd_1 and sd_2 split from n''s former subdomain, while updating SD (line 9). The master then informs both nodes of the split (line 10).

Finally, when no subdomain can be split (i. e. all non-completed subdomains contain only one point), the master stores the last tiles it receives (line 14) and sends a STOP signal to the workers (line 15).

3.4.2 Worker algorithm

We give the Subdomain worker algorithm in Algorithm 6. Each worker waits for messages from the master: whenever a STOP signal is received from m (m stands for the master node id), the worker terminates (line 3). Otherwise, a subdomain sd is received (line 4): the worker then covers sd with tiles (line 5 to line 12) by calling IM sequentially on consecutive integer points as in the Sequential (master) algorithm. The worker selects a point, sends it to the master for update purpose, calls IM on that point, sends the result to the master, asks for an update of the list of tiles, and so on. When sd is covered, the worker notifies

the master (line 13), and then waits again for a new message from the master until termination. Additionally, the worker checks whether the master has split its subdomain, because some other worker completed its own subdomain. In our implementation, this requires on the worker's side frequent (but inexpensive) checks whether the master has split the worker's current subdomain and, if so, a simple update of the subdomain.

Algorithm 6: Subdomain: Worker n

```
   input     : PTA 𝒜
   variables : Set of tiles T, point v_prev
1  while true do
2    switch receive() do
3      case m, STOP: do  return
4      case m, SUBDOMAIN(sd): do
5        while there are uncovered points in sd do
6          v ← Sequential.choosePoint()
7          send(m, NOTIFYPOINT(v))
8          K ← IM(𝒜, v)
9          send(m, RESULT(K))
10         send(m, REQTILES)
11         m, TILES(receivedTiles) ← receive()
12         T ← T ∪ receivedTiles
13       send(m, COMPLETED)
```

3.4.3 An additional heuristic

It may happen that, while a node is calling IM on a point v, another node has covered v with its own tile. For example, in Fig. 3.1a, node 2 calls IM on point v_2, while node 4 calls IM on point v_4. Assume calling IM on point v_2 yields K_2, that incidentally covers v_4. It is more efficient to stop the computation of IM on v_4, so that node 4 moves to another point instead of computing a redundant tile. We hence improve Subdomain by adding a heuristics that prevents this situation as follows: the master keeps track of all the points currently processed by each node; whenever a constraint computed by a node i covers the current node processed by another node j, the master informs immediately node j, and this node stops its computation to move to the next point. We refer to Subdomain augmented with this heuristics as Subdomain+H. This heuristics might be expensive, both on the master side and on the worker side (frequent checks to perform, and more communication), hence we will study both Subdomain and Subdomain+H.

3.5 Experiments

Experimental testbed description We ran our experiments on two clusters of Grid'5000: Pastel (located in Toulouse, France), and Griffon (located in Nancy, France). Pastel is made of 140 nodes, each of which features two dual-core AMD Opteron 2218 running at 2.6 GHz, 8 GiB of RAM and a GigaEthernet interconnection network. Griffon is made of 92 nodes, each of which features two quad-core Intel Xeon L5420 running at 2.5 GHz, 16 GiB of RAM and both GigaEthernet and 20G InfiniBand network interconnection networks. On these two clusters, the nodes were running a 64-bit Linux 3.2 kernel. The code was compiled using OCaml 4.01 and we used OpenMPI 1.8 with the OCamlMPI bindings.

We implemented our algorithms in IMITATOR [AFKS12] tool.[1]

Case studies We are presenting here results using seven case studies:

Flip-flop4 is a 4-parameter dimension asynchronous flip-flop circuit [CC04], made of four complex logical gates, and constrained by a predefined environment. Parameters are timing delays in the gate traversal delays, as well as setup and hold values for the input signals in the environment. Depending on the values of the parameters, the system can have a very different behavior.

RCP is a parametric model of the IEEE 1394 root contention protocol, where nodes must elect a leader. The model is inspired by the TREX [ABS01] model from the literature.

Sched3-2, Sched3B-2, Sched3B-3 and **Sched5** are parametric schedulability problems, where the goal is to find tiles where the system is robustly schedulable.

Sched3-2, Sched3B-2 and **Sched3B-3** are the same model, with a different number of parameters (2, 2 and 3 respectively), and a much larger D for Sched3B-2 and Sched3B-3, so as to test the scalability of our algorithms.

SiMoP is a parametric networked automation system, where several components communicate via a network bus [ACD+09].

We give in the "model" part of Table 3.2 the number of clocks, of parameters, and of integer points in D for each case study. In the "cartography" part, we give the number of tiles and the time (in seconds) to compute the non-distributed cartography ("monolithic"). Note that the number of tiles gives an upper bound on the number of nodes above which a perfect distribution algorithm cannot become more efficient: if each node computes a different tile, then using more than n nodes cannot be faster than n nodes. Hence, we bound the analysis to the smallest power of 2 greater or equal to # Tiles ("N_{max}").

[1] Sources, models and results are available at www.imitator.fr/static/ICFEM15/.

Case study	Flip-flop4	RCP	Sched3-2	Sched3B-2	Sched3B-3	Sched5	SiMoP	**Average**
Model								
Clocks	5	6	13	13	13	21	8	
Parameters	4	2	2	2	3	2	2	
$\|D\|$	386400	3050	286	14746	530856	1681	10201	
Cartography								
# Tiles	190	19	59	71	378	177	48	
N_{max}	128	32	64	128	128	128	64	
N for speedup	128	19	59	71	128	128	48	
Monolithic	1341.0	1992.0	46.0	61.2	865.0	3593.0	111.6	
Execution time at N_{max}								
Static	33.0	2108.0	4.0	26.6	181.0	213.0	21.4	
Sequential	2059.0	653.0	4.6	11.0	810.0	219.0	36.1	
Random	652.0	635.0	3.6	8.4	524.0	148.0	23.6	
Shuffle	670.0	624.0	3.1	7.6	243.0	140.0	18.7	
Subdomain	48.0	1286.0	7.2	15.8	217.0	273.0	32.4	
Subdomain+H	24.0	622.0	4.0	11.0	81.0	199.0	23.2	
Hybrid	24.0	624.0	3.1	7.6	81.0	140.0	18.7	
Ratio at N_{max} w.r.t. monolithic								
Static	2	106	9	43	21	6	19	29
Sequential	154	33	10	18	94	6	32	49
Random	49	32	8	14	61	4	21	27
Shuffle	50	31	7	12	28	4	17	21
Subdomain	4	65	16	26	25	8	29	24
Subdomain+H	2	31	9	18	9	6	21	14
Hybrid	2	31	7	12	9	4	17	12
Ratio at N_{max} w.r.t. slowest distr								
Static	2	100	15	40	22	6	21	29
Sequential	100	31	17	16	100	6	36	44
Random	32	30	14	13	65	4	23	26
Shuffle	33	30	12	11	30	4	18	20
Subdomain	2	61	27	24	27	8	32	26
Subdomain+H	1	30	15	16	10	6	23	14
Hybrid	1	30	12	11	10	4	18	12
Ratio at N_{max} w.r.t. slowest at N_{max}								
Static	2	100	56	100	22	78	59	60
Sequential	100	31	64	41	100	80	100	74
Random	32	30	50	32	65	54	65	47
Shuffle	33	30	43	29	30	51	52	38
Subdomain	2	61	100	59	27	100	90	63
Subdomain+H	1	30	56	41	10	73	64	39
Hybrid	1	30	43	29	10	51	52	31
Speedup at N_{max}								
Static	32	5	19	3	4	13	11	12
Sequential	1	16	17	8	1	13	6	9
Random	2	17	22	10	1	19	10	11
Shuffle	2	17	25	11	3	20	12	13
Subdomain	22	8	11	5	3	10	7	10
Subdomain+H	44	17	19	8	8	14	10	17
Hybrid	44	17	25	11	8	20	12	20

Table 3.2 – Complete summary of experiments

Methodology We compute BC for each algorithm, for a number of nodes from 4 to 128. For the sake of brevity, we study here the performances at $n = N_{max}$. The execution time (in seconds) is given in the third part ("Execution time") of Table 3.2. For the remaining parts of Table 3.2 and in the graphics of Figs. 3.8 to 3.10, we give also for each case study the execution time and the speedup the same number of nodes. Obviously, a low execution time is considered as good. The speedup is the execution time for an algorithm and a number of nodes N divided by the time needed for a perfect algorithm (i. e. the monolithic time divided by N). The speed-up is used to measure how the code scales, i. e. how much faster it runs as the number of nodes used for the computation increases. It is usually between 0 and 1. A high speedup (i. e. close to 1) is considered as good, while a value close to 0 denotes an inefficient algorithm (i. e. that does not scale). (The algorithm Hybrid will be explained later on.)

We use two metrics to evaluate our algorithms.

1. The first metric is the following ratio, that compares algorithms with each other, independently of their absolute performances: for each algorithm and each case study, we compute the time for this case study and this algorithm for N_{max} nodes divided by the maximum over all algorithms for this case study for N_{max} nodes, and multiplied by 100. A ratio equal to 100 means that this algorithm is the slowest for this case study, and a small ratio indicates a more efficient algorithm.

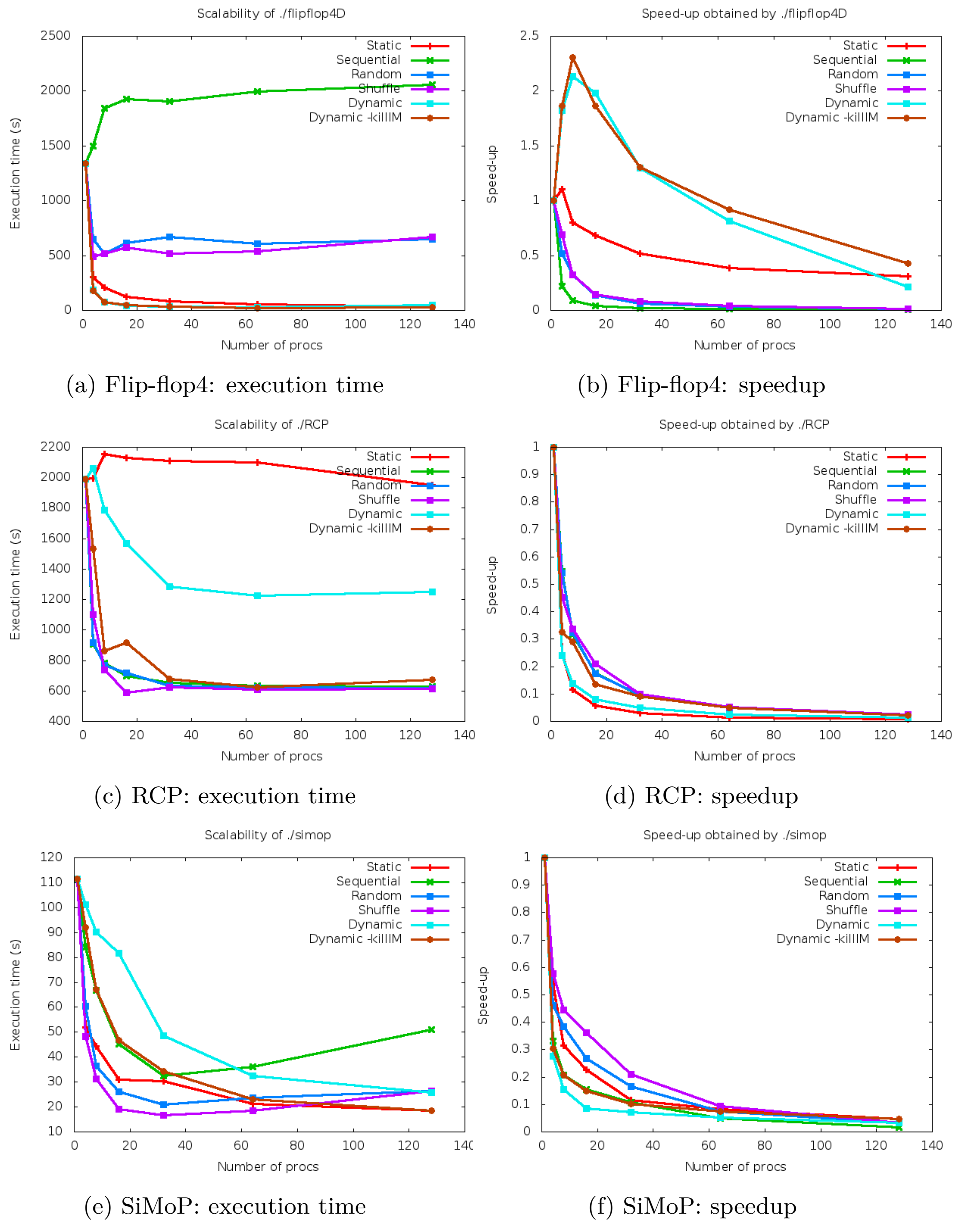
2. The second metrics is the speedup, that evaluates the scalability of each algorithm: for each algorithm and each case study, we compute the time for this case study and this algorithm for N_{max} nodes divided by the time needed for a perfect algorithm (i. e. the monolithic time divided by N_{max}), and multiplied by 100. Here, a number close to 100 means a very scalable algorithm, whereas a number close to 0 indicates an algorithm that does not scale well.

Finally, we explain here details of the remaining parts in table Table 3.2:

- the ratio at N_{max} w.r.t. the monolithic time, i. e. the execution time for an algorithm and a number of nodes N divided by the monolithic time and multiplied by 100 (of course, the smaller the better); note that a ratio greater than 100 means that the distributed algorithm is even slower than the monolithic one (which is the worst possible situation);

- the ratio at N_{max} w.r.t. the slowest distributed algorithm for any N, i. e. the execution time for an algorithm and a number of nodes N divided by the slowest distributed algorithm for any number of nodes and multiplied by 100 (again, of course, the smaller the better).

Note that Random is parameterized by the maximum number of attempts MAX before switching to a sequential enumeration. In all experiments, we used $MAX = 10$ (larger values did not significantly change the performances).



(a) Flip-flop4: execution time

(b) Flip-flop4: speedup

(c) RCP: execution time

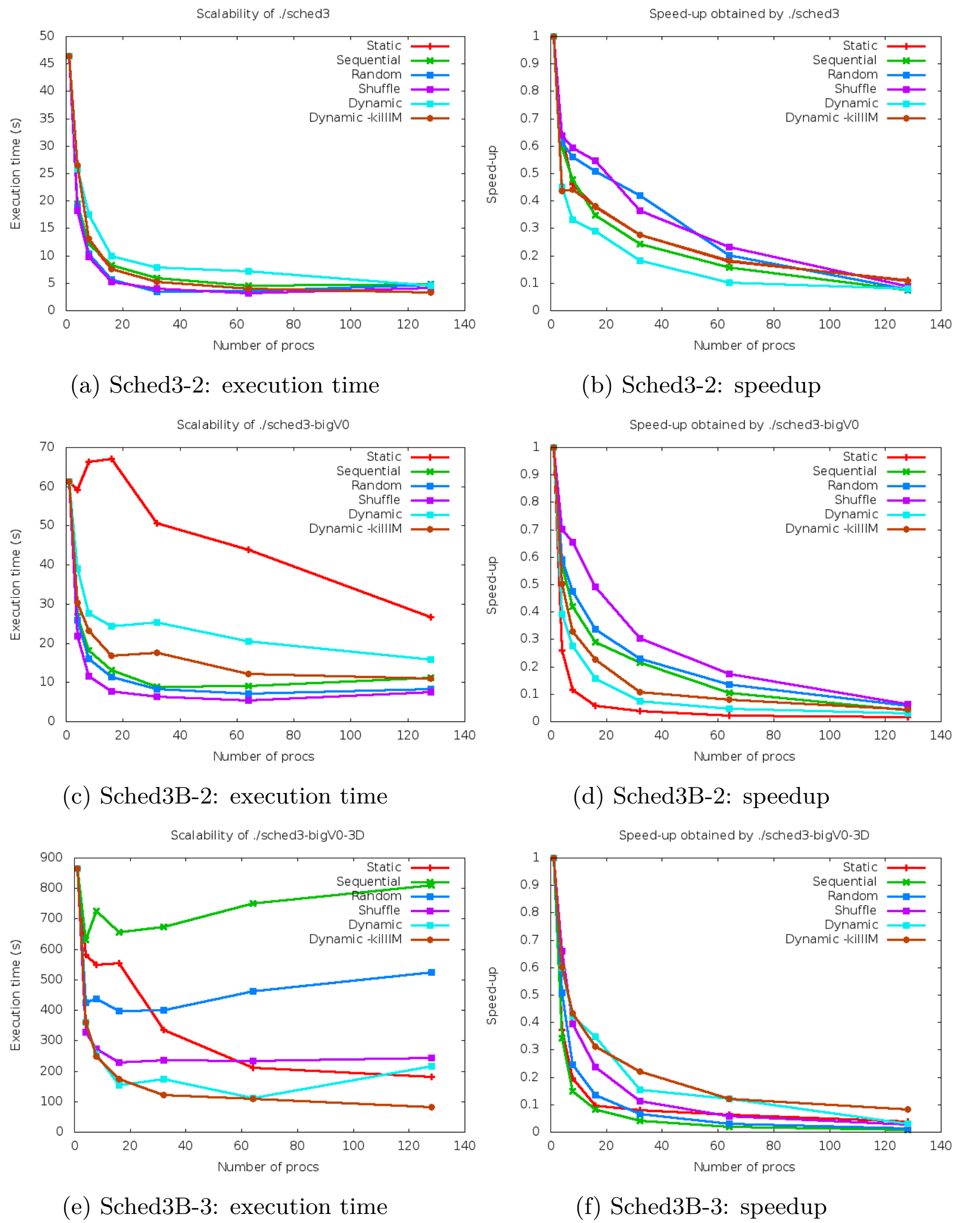
(d) RCP: speedup

(e) SiMoP: execution time

(f) SiMoP: speedup

Figure 3.8 – Experiments: execution time and speedup (1/3)

In the following, we describe the performance of each algorithm according to Table 3.2, before concluding which is the most efficient strategy.



(a) Sched3-2: execution time

(b) Sched3-2: speedup

(c) Sched3B-2: execution time

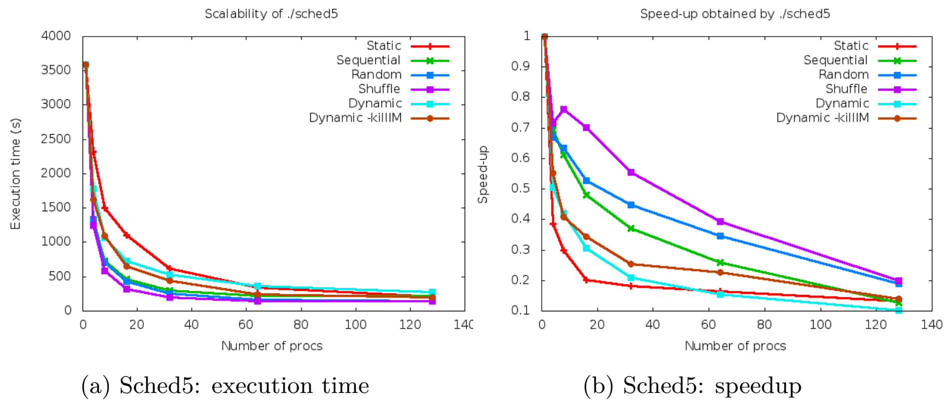
(d) Sched3B-2: speedup

(e) Sched3B-3: execution time

(f) Sched3B-3: speedup

Figure 3.9 – Experiments: execution time and speedup (2/3)

Static This static domain decomposition algorithm is clearly not efficient, and it shows that BC cannot be efficiently distributed using classical techniques for regular data distribution. Static is the worst algorithm twice (for RCP and



(a) Sched5: execution time

(b) Sched5: speedup

Figure 3.10 – Experiments: execution time and speedup (3/3)

Sched3B-2), and never the most efficient; a surprise is the very good performance for Flip-flop4, which probably comes from the fact that the tiles are very homogeneous geometrically for this case study, making a static distribution efficient.

Sequential Although it is easy to implement, this algorithm is terribly inefficient: with 3 case studies for which it is the worst algorithm, it is also the worst in average. This comes from the fact that Sequential is very likely to distribute to different nodes points that are close to each other, leading to redundant computations.

Random This algorithm behaves well for case studies with relatively few points in D, but it is always behind Shuffle in that case. It does not perform as well on case studies with large D, most likely because of the sequential enumeration of all points in the second phase of Random.

Shuffle With four case studies for which it is the best one, Shuffle is very efficient when D does not contain too many points. Shuffling the points guarantees a good random repartition of the points, without entailing complex operations at the master side... at the cost of being able to shuffle large quantities of points. This latter aspect certainly explains the low performances for Flip-flop4 and Sched3B-3.

Subdomain This algorithm is always outperformed by its variant Subdomain+H; it

seems that the cost of checking which node is computing which point and the additional necessary communications are largely compensated by the benefit of preventing redundant computations brought by stopping ongoing executions.

Subdomain+H This algorithm has the best average speedup (17 %). Although it clearly outperforms Shuffle for only two experiments (Flip-flop4 and Sched3B-3), Subdomain+H is for no case study very far from the best algorithm. This could make a good candidate for the best distribution algorithm – but we advocate in the following for a better proposition.

Conclusion: Hybrid From the experiments, we notice that Subdomain+H is always among the most efficient, but is outperformed by Shuffle for case studies with relatively few points in D. Hence, we propose the following "algorithm": if D contains relatively few points (say, less than 100,000), use Shuffle, otherwise use Subdomain+H. Note that the condition (number of points in D) only depends on the input of the analysis, and can be checked very easily. This new algorithm "Hybrid" is always the best one – except for RCP, for which it is very slightly slower than Subdomain+H despite a small number of points (3,050). In addition, Hybrid gets the smallest average ratio (31 %) and the highest speedup (20 %).

Discussion An average speedup of 20 % at N_{max} for Hybrid can seem relatively low; this means that a perfect distribution algorithm (that would always divide the monolithic computation time by N) would be 5 times faster. Still, we find it promising. First, all distributed algorithms suffer from the time spent in communication, which always lowers the speedup. Second, this confirms that distributing BC is far from trivial, due to the unknown shape of the cartography, the unknown computation time for each tile, and the risk for redundant computations. Third, and most importantly, a speedup of 20 % means that, when using 128 nodes, the computation time is still divided by more than 25 – which leads to an impressive decrease of the verification time.

3.6 Conclusion

We proposed in this chapter distribution algorithms to compute the cartography relying on the inverse method, along with presenting a heuristic approach to improve performance in parallelizing the inverse method approach among clusters.

In the next chapter, we will present another algorithm than IM to obtain different "cartographies"; this is the case of [ALNS15] where we use a reachability preservation algorithm ("PRP") instead of IM so as to obtain, not a behavioral cartography, but a simple "good/bad" partition with respect to a reachability property. We also reused the distribution schemes from this chapter for the PRP algorithm along with showing that PRP and its distributed algorithms always outperform IM algorithm.

CHAPTER 4

Reachability preservation based parameter synthesis for timed automata

Contents

4.1 Introduction **73**

4.2 Solving the EF-emptiness problem using reachability preservation **74**

4.2.1 Undecidability of the preservation of reachability . . 74

4.2.2 Parameter synthesis preserving the reachability . . . 76

4.2.3 EF-synthesis using PRP 79

4.3 Towards distributed parameter synthesis **80**

4.4 Experimental comparison **81**

4.5 Conclusion **84**

4.1 Introduction

In this chapter, our main goal is to address the EF-synthesis problem. Instead of attacking the state space exploration in a brute force manner (like [AHV93, JLR15]), we propose to perform several explorations of smaller size, taking advantage of reference valuations in the line of the inverse method.

Contributions In more details, our contributions are as follows:

1. We first address the following *reachability preservation problem* for PTA: given a reference parameter valuation v and a control state, do there exist other parameter valuations that reach this control state iff v does? We show that this problem is undecidable, and we introduce a procedure PRP (parametric reachability preservation) that gives a (possibly incomplete) answer.

2. Then, we show that PRP can efficiently replace IM in the behavioral cartography to partition a bounded parameter subspace into good and bad subdomains, and give a solution to the EF-synthesis problem.

3. We then compare the PRP-based cartography with the classical parameter synthesis semi-algorithm "EFsynth" [AHV93, JLR15] that solves the EF-synthesis problem: not only does PRP give a more precise result, but it also performs surprisingly well, despite its repeated analyses. Comparisons are performed using parametric schedulability problems for real-time systems.

4. We finally briefly discuss a distributed version of PRP, that is faster and almost always outperforms EFsynth.

Outline Section 4.2.1 defines the *reachability preservation problem* and proves its undecidability; Section 4.2.2 introduces PRP and proves its correctness; Section 4.2.3 shows that PRP can be used to solve the EF-synthesis problem. Section 4.3 discusses a distributed version of PRP, and Section 4.4 describes an experimental comparison with BC and EFsynth. Section 4.5 concludes the chapter and gives perspectives.

4.2 Solving the EF-emptiness problem using reachability preservation

4.2.1 Undecidability of the preservation of reachability

Parameter synthesis with respect to a bad location is known to be undecidable [AHV93]. Here, we take advantage of a reference parameter valuation v, for which it is possible to decide whether l_{bad} is reachable [AD94]. The assumption of a known parameter valuation seems realistic to us: in system design, it is often the case that one knows (from a previous design, of using empirical methods) a first valuation; however, finding other valuations may be much more difficult, and may require to restart the design phase from zero. Here, given a reference parameter valuation, we are interested in the preservation of the reachability of l_{bad} by other parameter valuations. Given two TA $\mathcal{A}[v]$ and $\mathcal{A}[v']$, we say that $\mathcal{A}[v']$ *preserves the reachability* of l_{bad} in $\mathcal{A}[v]$ when l_{bad} is reachable in $\mathcal{A}[v]$ if and only if l_{bad} is reachable in $\mathcal{A}[v']$. We call PREACH the problem of the preservation of reachability. In the following, we show that, given v, deciding

whether at least one parameter valuation $v' \neq v$ preserves the reachability of l_{bad} in $\mathcal{A}[v]$ is undecidable.

PREACH-emptiness problem:
INPUT: A PTA $\mathcal{A}$, and v a parameter valuation
PROBLEM: Does there exist $v' \neq v$ such that $\mathcal{A}[v']$ preserves the reachability of l_{bad} in $\mathcal{A}[v]$?

PREACH-synthesis problem:
INPUT: A PTA $\mathcal{A}$, and v a parameter valuation
PROBLEM: Compute the set of parameter valuations v' such that $\mathcal{A}[v']$ preserves the reachability of l_{bad} in $\mathcal{A}[v]$.

We show below that the PREACH-emptiness problem is undecidable.

Theorem 4.2.1. *PREACH-emptiness is undecidable.*

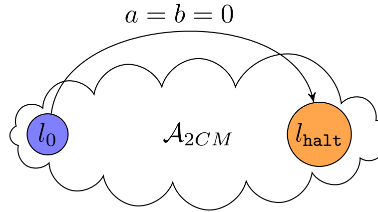


Figure 4.1 – Undecidability of PREACH-emptiness: PTA $\mathcal{A}$

Proof. Given a parameter valuation reaching some location, we reduce the existence of a different parameter valuation reaching the same location from the halting problem of a 2-counter machine.

1. First, recall that [AHV93] defines the encoding of a 2-counter machine (2CM) using a PTA $\mathcal{A}_{2CM}$ that contains two parameters a and b.[1] Then [AHV93] shows that the 2CM halts iff there exists at least one non-null parameter valuation such that a special location $l_{\texttt{halt}}$ is reachable in $\mathcal{A}_{2CM}$.

2. Now, let us add a gadget to $\mathcal{A}_{2CM}$ that adds a direct transition from the initial location l_0 to $l_{\texttt{halt}}$ with a guard $a = b = 0$.[2] Let $\mathcal{A}$ be this new PTA, as depicted in Fig. 4.1. Now, we have:

   (a) If the 2CM halts, then $l_{\texttt{halt}}$ is still reachable in $\mathcal{A}$ for some non-null parameter valuation since it was already reachable in $\mathcal{A}_{2CM}$. Additionally, due to our gadget, $l_{\texttt{halt}}$ is also reachable in $\mathcal{A}$ for $a = b = 0$.

[1] Strictly speaking, their construction uses six parameters, but it is well-known (shown, e. g. in [JLR15]) that they can be reduced to two.

[2] This guard is not allowed in PTA, but can be simulated using an extra clock x and an urgent location followed by a transition with guard $x = a \wedge x = b$.

(b) If the 2CM does not halt, $l_{\texttt{halt}}$ is again reachable in $\mathcal{A}$ for $a = b = 0$ due to our gadget, but no other parameter valuation can reach $l_{\texttt{halt}}$, just as in item 1.

Hence, given $v : a = b = 0$, there exists a parameter valuation $v' \neq v$ such that $\mathcal{A}[v']$ preserves the reachability of $l_{\texttt{halt}}$ in $\mathcal{A}[v]$ iff the 2CM halts.

□

4.2.2 Parameter synthesis preserving the reachability

To propose a solution to the PREACH-synthesis problem, we introduce here PRP$(\mathcal{A}, v)$, that is inspired by two existing algorithms, viz., EFsynth and the variant IM^K of IM [AS11] (see Appendix B.1.1 for the IM algorithm). Let us recall that in [AS11], the algorithm IM^K is obtained from IM by returning only the constraint K computed during the algorithm instead of the intersection of the constraints associated with all the reachable states. Thus, the constraint output by IM^K is weaker than the one outputted by IM and termination is the same for IM^K and IM. IM^K algorithm only prevents v-incompatible states to be reached but, contrarily to IM, does not guarantee that any "good" state will be reached. Therefore, this algorithm only preserves the non-reachability of locations.

Proposition 4.2.2. *Let $K_0 = \mathsf{IM}^K(\mathcal{A}, v_0)$. Then, for all $v \models K_0$, every trace of $\mathcal{A}[v]$ is equal to a trace of $\mathcal{A}[v_0]$.*

PRP (standing for parametric reachability preservation) is at first close to IM^K, and then switches to an algorithm that resembles EFsynth:

- As long as no bad location is reached, PRP generalizes the trace set of $\mathcal{A}[v]$ by removing v-incompatible states; this is done by negating v-incompatible inequalities, and returning the intersection of such negated inequalities, in the line of IM^K.
- When at least one bad location is met, PRP switches to an algorithm close to EFsynth, i. e. it simply gathers the constraints associated with the bad locations, and returns their union. However, a main difference with EFsynth is that PRP does not explore v-incompatible states: although this is not necessary to ensure correctness (in fact, this makes PRP not complete), this is a key heuristics to keep the state space of reasonable size.

We introduce PRP in Algorithm 7. It is a breadth-first exploration procedure that maintains the following variables: S (resp. S_{new}) is the set of states computed at the previous (resp. current) iterations; *Bad* is a Boolean flag that remembers whether a bad location has been met; K_{good} is the intersection of the negation of all v-incompatible inequalities, that will be returned if no bad state is met; K_{bad} is the union of the projection onto P of all bad states, that will be returned otherwise; i remembers the current exploration depth.

The procedure consists in a (potentially infinite) **while** loop. First, lines 3–7 take care of the v-incompatible states and resembles IM^K. These states are discarded from the exploration, i. e. they are removed from the set of new states (line 4). Then, if the exploration has not yet met any bad state, K_{good} is refined so as to prevent any such v-incompatible state (l, C) to be reached: a v-incompatible inequality J is selected within the projection of C onto P, and then its negation is added to K_{good}. This mechanism is borrowed to IM (and its variant IM^K).

Second, lines 8–9 take care of the bad states. If any bad state is reached (line 8), then the *Bad* flag is set to true, the union of the projection onto P of the constraints associated with these bad states is added to K_{bad}, and these states are discarded, i. e. their successor states will not be computed (line 9).

The third part is a classical fixpoint condition: if no new state has been met at this iteration (line 10), then the result is returned, i. e. either K_{bad} if some bad states have been met, or K_{good} otherwise. If new states have been met, then the procedure explores one step further in depth (line 12).

Algorithm 7: $\mathsf{PRP}(\mathcal{A}, v)$

input : PTA $\mathcal{A}$ of initial state s_0, parameter valuation v
output : Constraint over the parameters

```
1  S ← ∅ ;  S_new ← {s_0} ;  Bad ← false ;  K_good ← ⊤ ;  K_bad ← ⊥ ;
2  while true do
3      foreach v-incompatible state (l, C) in S_new do
4          S_new ← S_new \ {(l, C)}
5          if Bad = false then
6              Select a v-incompatible inequality J in C↓_P (i. e. s.t. v ⊭ J)
7              K_good ← K_good ∧ ¬J
8      foreach bad state (l_bad, C) in S_new do
9          Bad ← true ;  K_bad ← K_bad ∨ C↓_P ;  S_new ← S_new \ {(l_bad, C)}
10     if S_new ⊆ S then
11         if Bad = true then return K_bad  else return K_good ;
12     S ← S ∪ S_new ;  S_new ← Succ(S_new) ;  i ← i + 1
```

We will show in Theorem 4.2.4 that PRP outputs a sound (though possibly incomplete) answer to the PREACH-synthesis problem. In fact, PRP verifies a stronger property: if l_{bad} is reachable in $\mathcal{A}[v]$, PRP outputs a constraint K guaranteeing that l_{bad} is reachable for any parameter valuation satisfying K. However, if l_{bad} is unreachable in $\mathcal{A}[v]$, the constraint K output by PRP satisfies the same property as IM^K, i. e. the trace set of $\mathcal{A}[v']$ is a subset of the trace set of $\mathcal{A}[v]$, for all $v' \models K$. This is formalized in Proposition 4.2.3.

Proposition 4.2.3. *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. Suppose* PRP$(\mathcal{A}, v)$ *terminates with result K. Then, $v \models K$ and, for all $v' \models K$:*

- *if l_{bad} is reachable in $\mathcal{A}[v]$, then l_{bad} is reachable in $\mathcal{A}[v']$;*
- *if l_{bad} is unreachable in $\mathcal{A}[v]$, then every trace of $\mathcal{A}[v']$ is a trace of $\mathcal{A}[v]$.*

Proof. See Appendix C.2 □

Theorem 4.2.4. *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. Suppose* PRP$(\mathcal{A}, v)$ *terminates with result K. Then, $v \models K$ and, for all $v' \models K$, l_{bad} is reachable in $\mathcal{A}[v]$ iff l_{bad} is reachable in $\mathcal{A}[v']$.*

Proof. From Proposition 4.2.3. □

Remark. PRP may not terminate, which is natural since PREACH-synthesis problem in Section 4.2.1 is undecidable. Furthermore, even if it terminates, the result output by PRP may not be complete; in fact, this is designed on purpose (since we stop the exploration of v-incompatible states) so as to prevent a too large exploration. Enlarging the output constraint can be done by repeatedly calling PRP on other points than v, which will be done in Section 4.2.3.

Example 16. Let us apply PRP to the PTA $\mathcal{A}_1$ in Fig. 2.18. For point $v_1 : (a = 20, b = 10)$, PRP outputs constraint $20 > b \wedge a > b \wedge b \geq 0$, which guarantees the unreachability of l_{bad}. For point $v_2 : (a = 30, b = 20)$, PRP outputs constraint $b \geq 20 \wedge a \geq 0$, which guarantees the reachability of l_{bad}.

For point $v_3 : (a = 0, b = 40)$, PRP does not terminate.

We now state in Theorem 4.2.5 that, even when PRP is interrupted before its termination, PRP outputs a sound (though possibly incomplete) answer to the PREACH-synthesis problem, provided some bad states have already been met. The result comes from the fact that the first item of the proof of Proposition 4.2.3 holds even if PRP has not terminated. (Note that the converse case, when $Bad =$ `false`, does not hold if PRP has not terminated: although no bad state has been met yet, there could be some in the future.)

Theorem 4.2.5. *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. Let be K the value of K_{bad} at the end of iteration i of* PRP$(\mathcal{A}, v)$*, for some $i \geq 0$, such that $Bad =$ `true`. Then: 1) l_{bad} is reachable in $\mathcal{A}[v]$, and 2) for all $v' \models K$, l_{bad} is reachable in $\mathcal{A}[v']$.*

Example 17. Let us again apply PRP to the PTA $\mathcal{A}_1$ in Fig. 2.18. For this PTA and $v_3 : (a = 0, b = 40)$, PRP with a depth limit of 10 terminates with $Bad =$ `true`. From Theorem 4.2.5, the output constraint $(b \geq 20 \wedge a \geq 0) \vee (\wedge \geq 0 \wedge 9b \geq 20 \wedge b \geq a)$ is valid, i. e. guarantees the reachability of l_{bad}.

4.2.3 EF-synthesis using PRP

Given a bounded parameter domain, IM can be iterated on integer points to perform a behavioral cartography; then, the tiles can be partitioned in good and bad sets according to a linear-time property. If the property of interest is simply a (non-)reachability property, then PRP can be used in place of IM within BC, giving birth to a procedure PRPC (see Algorithm 8). PRP is called repeatedly with as an argument the first integer point not yet covered by any constraint (line 2 in Algorithm 8).

The "cartography" output by PRPC is less precise than the one output by the classical BC, because the constraints outputs by PRP are not tiles anymore: Theorem 4.2.4 only guarantees the preservation of reachability, and hence different parameter valuations within a constraint may correspond to different trace sets. To output a set of parameter valuations solving EF-synthesis, it suffices to return the union of the constraints for which l_{bad} is reachable.

Algorithm 8: PRPC($\mathcal{A}$,D)

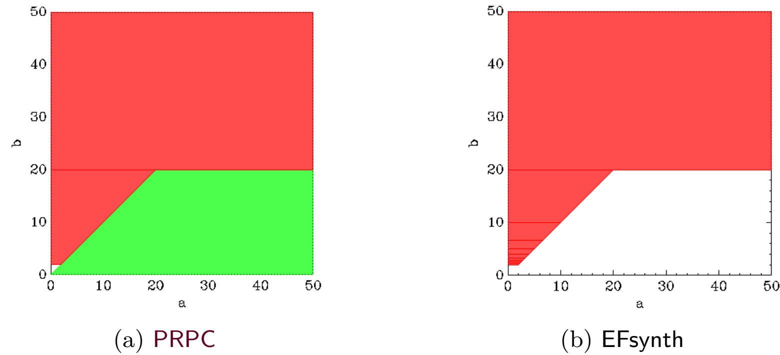
input : PTA $\mathcal{A}$, bounded parameter domain D
output : Set $\mathcal{C}$ of constraints over the parameters (initially empty)

```
1 while there are integer points in D not covered by C do
2 |   Select an integer point v in D not covered by C
3 |   C ← C ∪ PRP(A, v)
4 return C
```

Now, a key feature of PRPC is to explore a relatively small part of the whole parametric state space at a time, and to still output larger constraints than BC. We will show in Section 4.4 that using PRP instead of IM in the cartography indeed dramatically increases its efficiency.

Remark. In the general case, PRPC may not terminate, due to the non-termination of PRP. However, it is possible to set up a maximum exploration depth for PRP: when this depth is reached, the algorithm stops. If some bad states have been met, the resulting constraint can be safely used (from Theorem 4.2.5); otherwise the constraint is just discarded and the reference point on which PRP was called will never be covered. In this case, termination of PRPC is always guaranteed, with a partial result (some integer points may still be uncovered).

Let us now compare EFsynth and PRPC, that can both output (possibly incomplete) solutions to the EF-synthesis problem. On the one hand, EFsynth should be faster (although we will see in Section 4.4 that it is not even true in general), because it performs only one exploration, whereas PRPC has to launch PRP on many integer points. On the other hand, PRPC will use less memory, since a smaller part of the state space is explored at a time (due to the non-exploration of v-incompatible states). Furthermore, its main interest is that it synthesizes a more valuable result: whereas EFsynth outputs only a possibly under-approximated set of bad parameter valuations (reaching l_{bad}) and



(a) PRPC

(b) EFsynth

Figure 4.2 – EF-synthesis using PRPC and EFsynth for $\mathcal{A}_1$

leaves the whole rest of parameter valuations unknown, PRPC outputs possibly under-approximated sets of both bad and good parameter valuations, giving much more valuable information. Finally, just as BC, PRPC can possibly cover parameter valuations beyond the limits of D, which is not possible for EFsynth.

Example 18. Consider again the PTA $\mathcal{A}_1$ in Fig. 2.18, and let us apply EFsynth and PRPC with a bounded exploration depth of 10; recall that this is safe from Proposition 2.9.1 and Theorem 4.2.5. We apply PRPC to an unconstrained model with $D : a, b \in [0, 50]$. We apply EFsynth to a model where a and b are constrained to be in $[0, 50]$. We give in a graphical manner in Fig. 4.2a (resp. Fig. 4.2b) the results output by PRPC (resp. EFsynth). PRPC synthesizes all the good parameter valuations (below, in green), i. e. that do not reach l_2, and all the bad parameter valuations (above, in red), i. e. that reach l_2, with the exception of a small area near $(0, 0)$ (in white). All constraints output by PRPC are infinite (which is not shown in the figure), and hence cover the whole part outside D too. As of EFsynth, the same bad valuations as for PRPC are covered, but only within D , and no information is given about the good valuations. Hence, since EFsynth was stopped prematurely, no information can be given for the non-covered part: in particular, the white part of D cannot be decided, whereas PRPC covers everything except the small area near $(0, 0)$. This is a major advantage of PRPC over EFsynth in terms of precision of the result. Also recall that EFsynth covers only (a part of) D whereas PRPC covers here the whole parameter space beyond D .

4.3 Towards distributed parameter synthesis

In Chapter 3, we have shown that the **Sequential** and **Random** distribution algorithms are less efficient than the Subdomain distribution algorithm. Therefore, we will use Subdomain that dynamically splits the parametric domain D in subdomains: when a worker completes the covering of its subdomain, the master

splits another subdomain into two parts, and assigns one of the two part to that worker.

Remark (Fairness). Of course, comparing a distributed algorithm (PRPC) with a monolithic one (EFsynth) is unfair. However, to the best of our knowledge, no distributed algorithm for parameter synthesis has been proposed (except [ACE14]). One could argue that EFsynth could at least take advantage of multi-cores, e. g. using one core to compute the successor states while another performs the (costly) equality check, or by computing in parallel the successor states of several states – but PRPC could take advantage of exactly the same enhancements.

4.4 Experimental comparison

Description of the experimental testbed We compare here several algorithms to solve the EF-synthesis problem using IMITATOR [AFKS12]. IMITATOR implements EFsynth, BC and PRPC, and can run PRPC in a distributed fashion. Experiments were run on a Linux-based cluster. The nodes of this cluster feature two 6-core Intel Xeon X5670 running at 2.93 GHz CPUs (therefore, 12 cores in a NUMA fashion). Each node has 24 GiB of memory and runs a 64-bit Linux 3.2 kernel. The code was compiled using OCaml 3.12.1. The message-passing library we used is Bull's OpenMPI variant for Bullx, and the nodes are interconnected by a 40 Gb/s InfiniBand network.[3]

Case studies The case studies which we use in the experiment are described below.

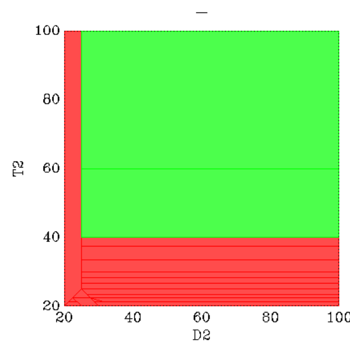
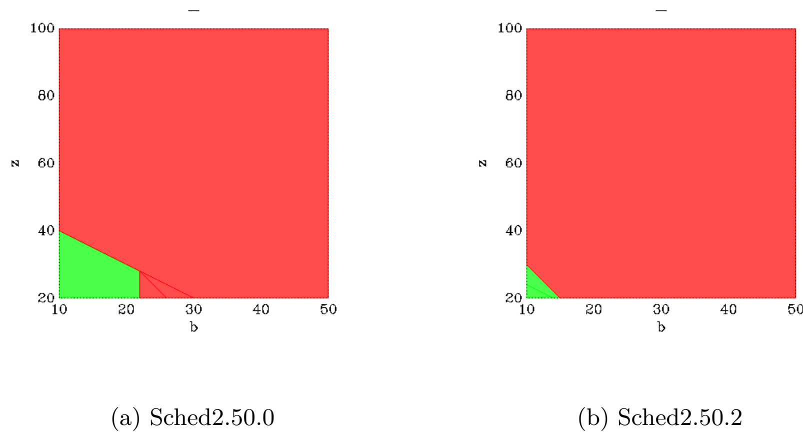


Figure 4.3 – Cartography output by PRPC for Sched1

[3]Sources, binaries, models and results are available at www.lipn.fr/~andre/PRP/.



(a) Sched2.50.0

(b) Sched2.50.2

Figure 4.4 – Cartography output by PRPC for Sched2 with $a = 50$

- **Dummy example**: At first, let us first study an example from [JLR15], shown in Figure 2.18. There are two clocks x and y, two parameters $a, b \in [0, 50]$ and l_2 is the *bad* location. The EFsynth method fails to terminate when synthesizing parameters in this example. By applying PRPC with at most 10 steps, we get the result in Figure 4.2. As can be seen, except for a small region being labeled as unknown, all other values of a and b are distinguished by PRPC as good (green) and bad (red).

- **Sched1** and **Sched2** are two parametric schedulability problems on a single processor. The goal is to synthesize task parameter valuations guaranteeing that every task meets its relative deadline.

  A real-time task τ_i is characterized by a tuple (C_i, D_i, T_i), where C_i is the execution, T_i is the period and D_i is the relative deadline. Every T_i time units the task releases a job, which must complete its execution C_i within D_i time units. Since D_i could be larger than T_i, a task may release a new job before the prior job finishes its execution, in which case the latter job is not able to execute till its precedence completes. Each task is assigned a unique and fixed priority. By convention, a lower task index corresponds to a higher priority.

  - **Sched1**: We consider two parameters D_2 and T_2 that correspond to the relative deadline and the period of task 2 respectively. A task set $\mathcal{T}$ with three tasks $\tau_1 = (5, 40, 40)$, $\tau_2 = (20, D_2, T_2)$ and $\tau_3 = (30, 100, 100)$, where $T_2, D_2 \in [20, 100]$ are parameters and other values are constants. We want to investigate the parameter space of T_2 and D_2 that guarantees all tasks meet their deadlines. The graphical cartography output by PRP for Sched1 is given in Fig. 4.3.

- **Sched2**: We consider here a scheduling example, adapted from the example studied in [BFSV04, JLR15]. There are three tasks τ_1, τ_2 and τ_3: $D_1 = T_1 = a$ and $C_1 \in [10, b]$; $D_2 = T_2 = 2a$ and $C_2 \in [18, 28]$; $D_3 = T_3 = 3a$ and $C_3 \in [20, z]$. Moreover, task τ_2 has a release jitter $J_2 \in \{0, 2\}$. Jitter reflects the uncertainty of task activation. For example, the time interval between two successive job releases of task τ_2 can be any value in $[T_2, T_2 + J_2]$. When modeling the scheduling, for each task there is a clock to track the time passed since its latest job release and to trigger the next task activation. We suppose such clocks are initialized to 0. In the following experiments, we will consider b and z as parameters. The graphical cartographies output by PRP for Sched2.50.0 and Sched2.50.2 are given in Fig. 4.4.
  We will study Sched2 with two different D. First, we valuate $a = 50$, we set $D : b \in [10, 50], z \in [20, 100]$ and we synthesize parameters for both $J_2 = 0$ ("Sched2.50.0") and $J_2 = 2$ ("Sched2.50.2"). Second, we valuate $a = 100$, we set $D : b \in [10, 1000], z \in [20, 1000]$ and we consider $J_2 = 0$ ("Sched2.100.0") and $J_2 = 2$ ("Sched2.100.2").

- **Sched5**: It models the schedulability of 5 fixed-priority tasks in a single processor. SPSMALL is a model of an asynchronous memory circuit [CETFX09].

Methodology Table 4.1 gives from left to right the case study, the number of clocks, the number of integer points in D and the computation time in seconds for EFsynth, BC, PRPC, and the distributed version of PRPC using the part-splitting point distribution running on 12 nodes. "TO" indicates a timeout ($> 5,000\,s$).

Case study	$\lvert H\rvert$	$\lvert V\rvert$	EFsynth	BC	PRPC	PRPC distr(12)
$\mathcal{A}_1$	2	2,601	0.401*	TO	0.078*	0.050*
Sched1	13	6,561	TO	TO	1595	219
Sched2.50.0	6	3,321	9.25	990	14.55	4.77
Sched2.50.2	6	3,321	662	TO	213	84
Sched2.100.0	6	972,971	21.4	2093	116	10.1
Sched2.100.2	6	972,971	3757	TO	4557	1543
Sched5	21	1,681	352	TO	TO	917
SPSMALL	11	3,082	7.49	587	118	11.2

Table 4.1 – Comparison of algorithms to solve the EF-synthesis problem

For $\mathcal{A}_1$, none of the algorithms terminate; hence, termination is ensured by bounding the exploration depth to 10 (marked with * in Table 4.1). From Proposition 2.9.1 and Theorem 4.2.5, the result is still correct; however, this does not hold for BC. For the other case studies, all algorithms terminate (except in case of timeouts), and always cover entirely D. To allow a fair comparison, parameters for EFsynth are bounded in the model as in D ; without these bounds, EFsynth never terminates for these case studies.

First, we see that PRPC dramatically outperforms BC for all case studies. This is due to the fact that the constraints output by PRP (that preserve only

non-reachability) are much weaker than those output by IM (that preserve trace set equality). Second, we see that PRPC compares rather well with EFsynth, and is faster on three case studies; PRPC furthermore outputs a more valuable constraint for $\mathcal{A}_1$ (see Example 18). PRPC can even verify case studies that EFsynth cannot (Sched1).

Conclusion The distributed version of PRPC is faster than PRPC for all case studies. Most importantly, the distributed PRPC outperforms EFsynth for all case studies but two. The good timing efficiency of PRPC is somehow surprising, since it was devised to output a more precise result and to use less memory, but not necessarily to be faster. We believe that PRPC allows to explore small state spaces at a time and, despite the repeated executions, this is less costly than handling a large state space (as in EFsynth), especially when performing equality checks when a new state is computed.

4.5 Conclusion

In this chapter, we addressed the synthesis of timing parameters for reachability properties. We introduced PRP that outputs an answer to the parameter synthesis problem of the preservation of the reachability of some bad control state l_{bad}, which we showed to be undecidable. By repeatedly iterating PRP on some (integer) points, one can cover a bounded parameter domain with constraints guaranteeing either the reachability or the non-reachability of l_{bad}. This approach competes well in terms of efficiency with the classical bad state synthesis EFsynth, and gives a more precise result than EFsynth while using less memory. Finally, our distributed version almost always outperforms EFsynth and distributing PRP using Subdomain often outperforms the monolithic bad-state reachability synthesis (e. g. [AHV93, JLR15]). Hence, we believe that our point distribution algorithms can be reused for different purposes than just BC.

Unfortunately, even though these parameter synthesis algorithms are distributed on clusters, they still suffer from the state space explosion. In order to go further, in Chapter 5, we present first the parametric zone inclusion algorithm, an extension of the well-known zone inclusion algorithm of timed automata. It is not optimal and this causes exploring unnecessary locations. Indeed, the zone inclusion algorithm is very sensitive to the state exploration order. Therefore, we will show how we can improve efficiency of the parametric zone inclusion algorithm by using our variety state exploration order strategies.

CHAPTER 5

Efficient parameter synthesis using optimized state exploration strategies

Contents

5.1 Introduction **85**
5.2 Parametric zone inclusion algorithm **87**
5.3 Parametric ranking strategy **89**
5.4 Parametric priority strategy **91**
5.5 Experimental evaluation **94**
5.5.1 Symbolic state merging 95
5.5.2 Comparison 95
5.5.3 Final interpretation 96
5.6 Conclusion **99**

5.1 Introduction

Parameter synthesis algorithms usually rely on the *parametric zone graph* (a parametric extension of the zone graph of TAs [BY03] which is recalled in Section 2.5.4) where states are pairs consisting of a discrete location and a parametric zone describing the set of possible parameter and clock valuations

in this state (see e. g. [HRSV02, AS13, JLR15]). The parametric zone graph is not only subject to the well-known state space explosion problem, but is usually even infinite. Indeed, Section 2.8.2 shows that most problems for PTAs are undecidable, including the emptiness of the parameter valuation set for which a given location is reachable ("EF-emptiness problem") [AHV93].

Depth-first search (DFS) and breadth-first search (BFS) are popular exploration orders of model checking algorithms. By observation in practice, many authors (e. g. [BHV00, Beh05]) showed that using BFS is much more efficient than DFS for checking reachability properties in TAs.

In [HT15], the authors show that in some cases, BFS can explore an exponential number of unnecessary states in TAs: this happens when a zone is found after another state with a strictly smaller zone (and the same discrete location) is found. We call this state with smaller zone a *redundant state* and this phenomenon an *inefficient phenomenon*.

Contribution We study here various exploration orders to address efficient parameter synthesis for PTAs. By taking a different exploration to reach the larger zone first, we propose two main exploration strategies to reduce the inefficient phenomenon and increase the efficiency of parameter synthesis in PTAs. Our contributions are as follows:

1. The first exploration order we propose is a *parametric ranking strategy*, inspired by the ranking system of [HT15], based on the ranking of each explored parametric zone, then to stop the exploration of a small zone and its subtree when a larger zone of the same location is explored later.

2. The second is a *parametric priority strategy*, which explores the biggest zone first in order to avoid the inefficient phenomenon and then stop the exploration from a small zone by correcting the inefficient phenomenon automatically.

3. We also compare these exploration orders with the classical BFS strategy, and the layer BFS LayerBFS, which is a variant of BFS historically implemented in IMITATOR. We perform extensive experiments using the IMITATOR software [AFKS12] that takes as input parametric timed automata. First, we show that our new strategies always outperform the BFS strategy. Second, when using an additional existing state space optimization called "convex state merging" [AFS13a] (that can be used only for reachability properties), BFS becomes best again. However, for counter-example synthesis (i. e. try to find *some* parameter valuations instead of all), our exploration strategies significantly outperform BFS, with an average speed-up of 5.

Note that, in the worst case, these our two strategies explore unnecessary visited parametric zones exponentially.

Related works As noted in [HT15], the exploration order problem was addressed in the context of state caching focusing on limiting the number of stored nodes at nodes exploring cost [GHP92, BLP03, EK10], and state space fragmentation [DT98, BHV00, BOS02, Beh05]. In [BF01], a value has been added to guide the exploration in priced timed automata, which has been reused in [HT15], and that we reuse in our second exploration strategy. Also note that the exploration order was considered in several works in the framework of *distributed* model checking for TAs [BHV00, BOS02, Beh05], where it seems that BFS is the most optimal exploration order. Zone inclusion was also considered in [LOD+13] in multi-core model checking of TAs. To the best of our knowledge, comparing exploration strategies was never considered for PTAs or more generally for parametric timed formalisms. While we partially rely on exploration strategies for TAs, the differences of data structures (DBMs [BY03] cannot be used in PTAs but some extensions of it can, such as PDBM [HRSV02] and CPDBM [BBBC16]) and the specificities of the symbolic zones for PTAs (that include not only clock valuations but also parameter valuations) make it important to study these strategies for PTAs.

Outline We first recall the parametric zone inclusion algorithm for parametric timed automata in Section 5.2. Then, we introduce in Sections 5.3 and 5.4, parametric ranking strategy and parametric priority strategy respectively to limit the inefficient phenomenon during exploration. Section 5.5 provides experimental results of our approaches. Finally, we conclude in Section 5.6.

5.2 Parametric zone inclusion algorithm

Similar to timed automata's zone inclusion, *parametric zone inclusion* is an optimization technique relying on the parametric zone graph. That is, for some properties (including reachability and safety), given two reachable states $\mathbf{s}_1 = (l_1, C_1)$ and $\mathbf{s}_2 = (l_2, C_2)$, whenever $l_1 = l_2$ and $C_1 \subseteq C_2$, it is safe to replace $\mathbf{s}_1$ with $\mathbf{s}_2$ in the analysis. This inclusion check is even more costly in PTAs than its counterparts in TAs, but it is usually compensated by the performance improvement obtained by the decrease of the number of symbolic states to consider.

Algorithm 9 describes the standard state exploration algorithm with zone inclusion for PTA. It explores the infinite abstract parametric zone graph of PTA $\mathcal{A}$ from its initial location. The intuition of parametric zone inclusion is to stop the exploration of a small zone whenever a larger zone with the same location is explored. Therefore, in order to look up information of visited states having smaller zones at a certain location and do the zone inclusion, Algorithm 9 maintains a set of waiting states $\mathcal{W}$ and a graph $\mathcal{G}$ containing both visited states and transitions between them. In Algorithm 9, two situations can lead to zone inclusion.

The first is the classical one, at line 14: if a large zone has already been explored earlier, it subsumes the smaller zone being explored, which will be

Algorithm 9: State exploration with parametric zone inclusion

```
   Input: PTA 𝒜 = (Σ, L, l₀, X, P, K₀, I, E)
   Output: parametric zone graph 𝒵 associated with the PTA 𝒜
 1 𝒲 ← {(l₀, C₀)}
 2 𝒢 ← {(l₀, C₀)}
 3 while 𝒲 ≠ ∅ do
 4     pick a state (l, C) from 𝒲
 5     foreach outgoing state (l′, C′) from (l, C) do
 6         if there is no (l′, C_Larger) ∈ 𝒢 such that C′ ⊆ C_Larger then
 7             add (l′, C′) to 𝒲 and 𝒢
 8             add transition (l, C) → (l′, C′) to 𝒢
 9             foreach (l′, C_Smaller) ∈ 𝒢 such that C_Smaller ⊆ C′ do
10                 add transitions to 𝒢:
11                 parent states of (l′, C_Smaller) → (l′, C′) and
12                                       (l′, C′) → children states of
                   (l′, C_Smaller)  remove (l′, C_Smaller) from 𝒲 and 𝒢
13         else
14             foreach (l′, C_Larger) ∈ 𝒢 such that C′ ⊆ C_Larger do  add
                 transition (l, C) → (l′, C_Larger) to 𝒢
15 return 𝒢
```

included by the larger zone with same location. Thus, only a transition from (l, C) to the larger zone (l', C_{Larger}) is added to $\mathcal{G}$, and not the newly computed state with a smaller zone (l', C').

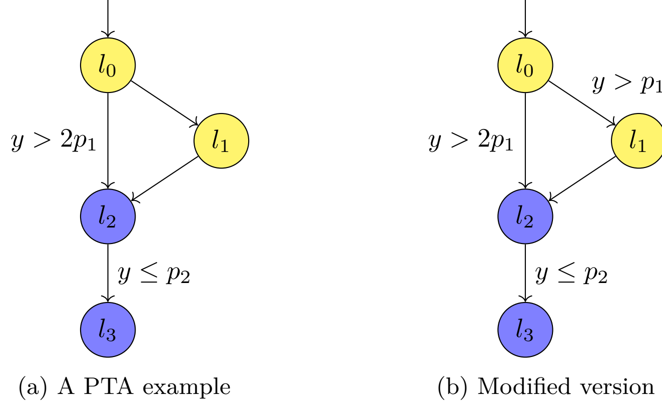
The second situation where the inefficient phenomenon happens is when a larger zone is explored *after* exploring smaller zones. At line 9, the algorithm looks for the previous smaller parametric zones in the set of visited states $\mathcal{G}$, then removes states with smaller zones with its incoming and outgoing transitions. Also, transitions from parents of the smaller zones to the bigger zone and from the bigger zone to the children of the smaller zones are added at lines 11 and 12.

Note that, by exploring the bigger zones, the smaller zones and their successors or subtrees will eventually be pruned. Within the second situation, the parametric zone inclusion algorithm stores fewer nodes (i. e. symbolic states of the parametric zone graph) but this overhead of smaller zone removal procedure slightly influences the performance of the parametric zone inclusion algorithm.

Experiments will be reported in Section 5.5, where the performances of the different inclusions will be compared.

Note that Algorithm 9 does not mention any exploration order for any specific property checking. Choosing the exploration order will affect the performance of the algorithm, inefficient phenomenon and number of nodes visited by the algorithm and stored in the sets $\mathcal{W}$ and $\mathcal{G}$.

Example 19. We reuse in Fig. 5.1a a part of a parameterized version of the



(a) A PTA example

(b) Modified version

Figure 5.1 – Examples

FDDI case study of [HT15]. Fig. 5.1a depicts an example of PTA with two clocks x and y (x does not appear on any guard nor invariant) and two parameters p_1 and p_2. Action labels are not shown. The transition from l_0 to l_2 is guarded by $y > 2p_1$ while the transition from l_2 to l_3 is guarded by $y \leq p_2$. Let us consider two different exploration strategies. The first parametric zone graph in Fig. 5.2a is explored by the standard BFS exploration order and the one in Fig. 5.2c by BFS with the parametric zone inclusion. Here we can see that by using parametric zone inclusion, the number of states to be explored is often reduced. Let us explain how Algorithm 9 works on the example in Fig. 5.1a using Fig. 5.2c. The algorithm starts at state $\mathbf{s}_0 : (l_0, \mathit{true})$, the location is l_0 and the parametric zone is *true* (i. e. the set of all clock and parameter valuations). Assume that the transition to l_2 is taken first. The algorithm reaches states $\mathbf{s}_1 : (l_2, y > 2p_1)$ and $\mathbf{s}_2 : (l_1, \mathit{true})$. Later on, the algorithm reaches states $\mathbf{s}_3 : (l_3, 2p_1 < y \leq p_2)$ and $\mathbf{s}_4 : (l_2, \mathit{true})$. At that stage, it happens that the parametric zone in $\mathbf{s}_4 : (l_2, \mathit{true})$ is larger than the parametric zone in $\mathbf{s}_1 : (l_2, y > 2p_1)$ which has been visited previously. This previous exploration turns out to be useless, hence state $\mathbf{s}_1$ is removed and a transition from $\mathbf{s}_0$ to $\mathbf{s}_4$ is added. Finally, the algorithm does the same with states $\mathbf{s}_5$ and $\mathbf{s}_3$.

The ideal exploration is depicted in Fig. 5.2d. If the algorithm takes first the transition to location l_1 and then to l_2, the result is optimal. The goal of the remaining of this this chapter will be to get as close as possible to this optimal exploration order so as to avoid *redundant states*.

5.3 Parametric ranking strategy

In this section, we propose a novel exploration strategy for PTAs, inspired by the "ranking system" strategy that proved efficient for reducing the inefficient phenomenon in TAs [HT15]. As in [HT15], our parametric ranking strategy

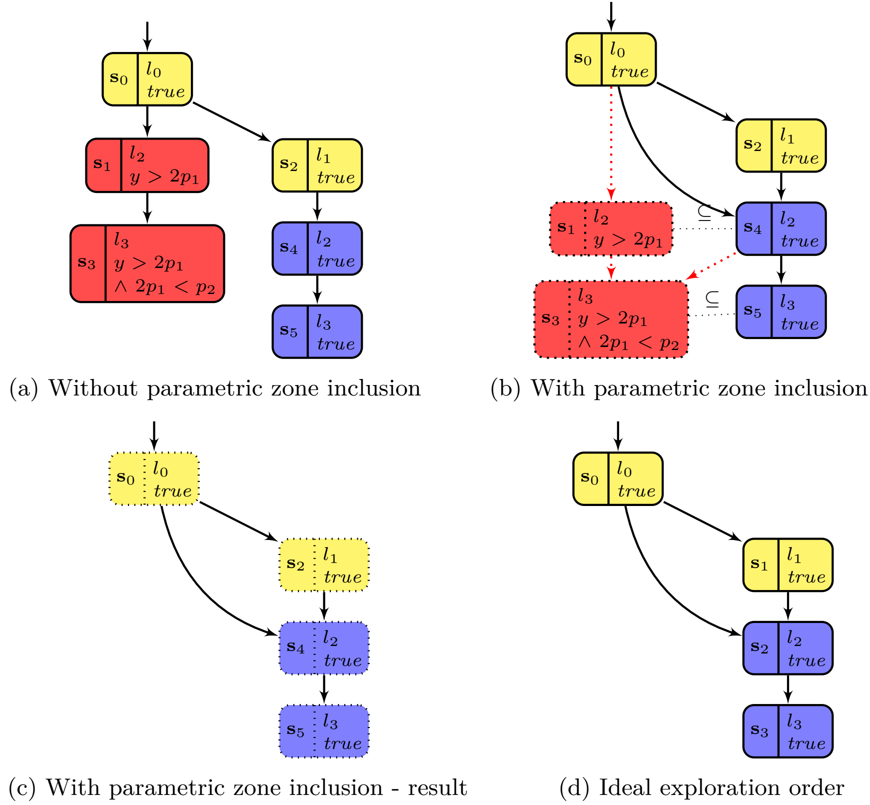


Figure 5.2 – Parametric zone graphs of Fig. 5.1a where the number n in a state label $\mathbf{s}_n$ reflects the exploration order

uses a priority value for each state. Then the algorithm explores the state with highest priority first. In case the inefficient phenomenon happens, the larger zone is assigned a higher priority than the smaller parametric zone and its previously explored subtrees.

The parametric ranking strategy in Algorithm 10 is an extension of the parametric zone inclusion of Algorithm 9 (differences are highlighted). Each newly explored state starts with being ranked with infinity (if its constraint is *true*) or zero (otherwise) by Algorithm 11. Note that, different from TAs, the initial constraint in PTAs is often not *true* but a constraint over $X \cup P$ that also contains parameter constraints (for example $p_1 \leq p_2$). We assume w.l.o.g. that *true* denotes this initial constraint (for example $p_1 \leq p_2$).

At lines 9 and 10, in order to stop the exploration of small parametric zones and their subtrees, the rank of the larger parametric zone is set higher than the highest rank of the small parametric zone and those in its subtree. This procedure is described in Algorithm 12: at line 3 it traverses all visited descendants of

Algorithm 10: Ranking by parametric zone size

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, PK_0, K_0, I, E)$

Output: parametric zone graph $\mathcal{Z}$ associated with the PTA $\mathcal{A}$

1. $r_0 \leftarrow init_rank(l_0, C_0)$
2. $\mathcal{W} \leftarrow \{((l_0, C_0), r_0)\}$
3. $\mathcal{G} \leftarrow \{(l_0, C_0)\}$
4. **while** $\mathcal{W} \neq \emptyset$ **do**
5. pick a state $((l, C), r)$ with highest rank r from $\mathcal{W}$
6. **foreach** *outgoing state* (l', C') *from* (l, C) **do**
7. $r' \leftarrow init_rank(l', C')$
8. **if** *there is no* $((l', C_{Larger}), r_L) \in \mathcal{G}$ *such that* $C' \subseteq C_{Larger}$ **then**
9. **foreach** $((l', C_{Smaller}), r_S) \in \mathcal{G}$ *such that* $C_{Smaller} \subseteq C'$ **do**
10. $r' \leftarrow \max(r', max_rank((l', C_{Smaller}), r_S) + 1)$
11. add $((l', C'), r')$ to $\mathcal{W}$ and $\mathcal{G}$
12. add transition $((l, C), r) \rightarrow ((l', C'), r')$ to $\mathcal{G}$
13. **foreach** $((l', C_{Smaller}), r_S) \in \mathcal{G}$ *such that* $C_{Smaller} \subseteq C'$ **do**
14. add transitions to $\mathcal{G}$:
15. parent nodes of $((l', C_{Smaller}), r_S) \rightarrow ((l', C'), r')$ and
16. $((l', C'), r') \rightarrow$ children states of $((l', C_{Smaller}), r_S)$
17. remove $((l', C_{Smaller}), r_S)$ from $\mathcal{W}$ and $\mathcal{G}$
18. **else**
19. **foreach** $((l', C_{Larger}), r_L) \in \mathcal{G}$ *such that* $C' \subseteq C_{Larger}$ **do**
20. add transition $((l, C), r) \rightarrow ((l', C_{Larger}), r_L)$ to $\mathcal{G}$
21. **return** $\mathcal{G}$ *(without rank values)*

Algorithm 11: *init_rank*(l, C)

1 **if** $C = true$ **then return** ∞ **else return** 0

Algorithm 12: *max_rank*$((l, C), r)$

Output: *rank* value

1 $rank \leftarrow r$
2 **if** $((l, C), r) \notin \mathcal{W}$ **then**
3 **foreach** $((l, C), r) \rightarrow ((l', C'), r')$ *in* $\mathcal{G}$ **do**
4 $rank \leftarrow \max(rank, max_rank((l', C'), r'))$
5 **return** *rank*

$(l', C_{Smaller})$ to get their highest rank. Since the larger zone has a higher rank, it will be explored before the smaller ones and their subtrees.

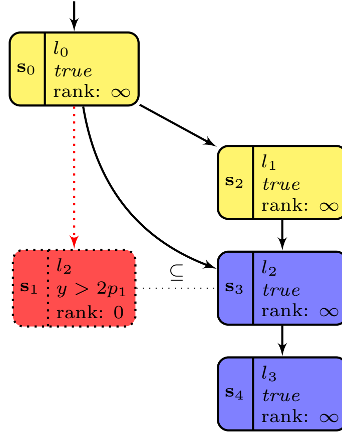


Figure 5.3 – PZG with parametric ranking strategy

Example 20. Let us apply the parametric ranking strategy of Algorithm 10 to the example in Fig. 5.1a. The resulting parametric zone graph is shown in Fig. 5.3. Starting at state $(l_0, true)$, the algorithm ranks it with ∞. Then, states $\mathbf{s}_1 : (l_2, y > 2p_1)$ and $\mathbf{s}_2 : (l_1, true)$ are explored and added in the waiting set $\mathcal{W}$ with rank 0 and ∞ respectively. Hence, $\mathbf{s}_2$ with rank ∞ is explored first and leads to state $\mathbf{s}_3 : (l_2, true)$. At that stage, the algorithm detects that the zone of $\mathbf{s}_1 = (l_2, y > 2p_1)$ is smaller than that of $\mathbf{s}_3 = (l_2, true)$. The rank of $\mathbf{s}_3$ is ∞. The predecessor of $\mathbf{s}_1$ (i. e. $\mathbf{s}_0$) is connected to $\mathbf{s}_3$ and $\mathbf{s}_1$ is deleted. Finally, $\mathbf{s}_4$

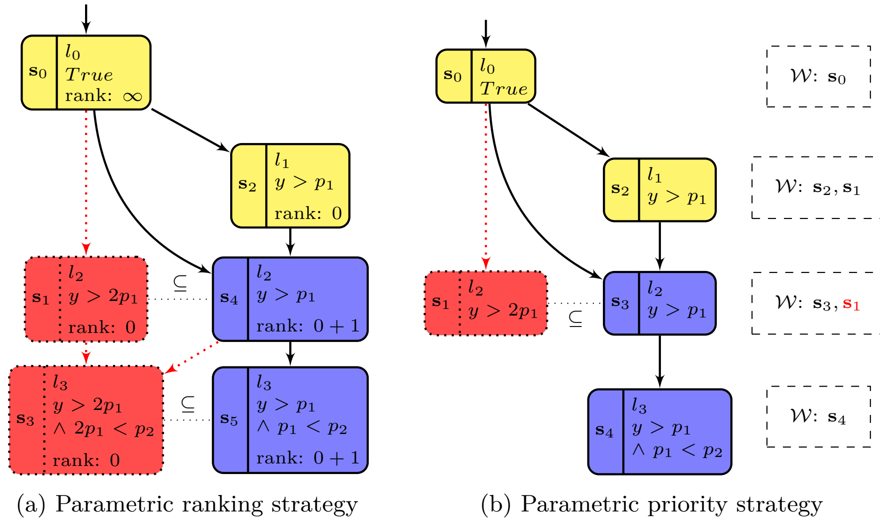
(the successor of $\mathbf{s}_3$) is added with rank 0.

5.4 Parametric priority strategy

In [HT15], the authors indicate that with the "ranking system", there is no improvement if there are no *true* zones in a model, compared to using the BFS exploration order. The same holds for our "parametric ranking" strategy. Indeed, first assigning the highest and lowest priority to each state, and then looking for visited states in order to find the highest rank in the subtrees (in large models where the subtrees become big) might be not efficient. Consider the example in Fig. 5.1a, where the guard of the transition from l_0 to l_1 is modified as in Fig. 5.1b. Its parametric zone graph is given in Fig. 5.4a.

In Fig. 5.4a, the parametric ranking algorithm ranks states from $\mathbf{s}_0$ to $\mathbf{s}_3$ with 0 continuously. The inefficient phenomenon is encountered as the parametric zone of $\mathbf{s}_4$ is larger than the one of $\mathbf{s}_1$ with the same location l_2. In this case, $\mathbf{s}_3$ is explored unnecessarily, similar to using the BFS exploration order.

However, after reaching states $\mathbf{s}_1$ and $\mathbf{s}_2$, if the algorithm explores the largest parametric zone first, i. e. $\mathbf{s}_2$, then $\mathbf{s}_4$ is reached earlier.



(a) Parametric ranking strategy

(b) Parametric priority strategy

Figure 5.4 – Comparing our two strategies

Hence, to avoid this inefficient phenomenon, we introduce a new strategy that explores the largest zone first in the sorted waiting list $\mathcal{W}$. Furthermore, in order to avoid traversing big subtrees to find the highest rank, we explore the largest zones until there is no inefficient phenomenon anymore. To do so, we use a simple inserting mechanism.

Before getting into the details of Algorithm 13, we explain the structure of the waiting list $\mathcal{W}$. First, $\mathcal{W}$ is ordered with decreasing zones. Hence there are

two main parts in $\mathcal{W}$, the first (at the head) is the true zones part where all true zones are located. The other is the non-true zone part. Finally, since some non-true zones are incomparable, the true zones part can be seen as being itself composed of several parts each containing ordered comparable zones.

In Algorithm 13, the waiting list $\mathcal{W}$ described above, is sorted from largest to smallest zone by the inserting instructions from line 14 to line 19.

There are three possibilities. First, if C' is the true zone (which has the highest priority), (l', C') is inserted at the beginning of list $\mathcal{W}$. Next, if C' is not a true zone, (l', C') is inserted before the first smaller zone found in $\mathcal{W}$. Finally, in case all zones in $\mathcal{W}$ are incomparable with C', (l', C') is added at the end of list $\mathcal{W}$.

For better performances, the implementation uses an additional index (not described in Algorithm 13) storing information on the sets of comparable zones, for faster state insertion and comparison by avoiding repeated zone constraint computation.

Example 21. Let us apply Algorithm 13 to the example in Fig. 5.1b. Fig. 5.4b shows both its parametric zone graph and the waiting list $\mathcal{W}$. The algorithm starts with state $\mathbf{s}_0$ in waiting list $\mathcal{W}$ and reaches the states $\mathbf{s}_1$ and $\mathbf{s}_2$ which are inserted into $\mathcal{W}$ decreasingly. Because the zone $(y > 2p_1)$ in $\mathbf{s}_1$ is smaller than the zone $(y > p_1)$ in $\mathbf{s}_2$ then $\mathbf{s}_2$ appears before $\mathbf{s}_1$ in $\mathcal{W}$. Then, the algorithm picks $\mathbf{s}_2$ from the head of list $\mathcal{W}$ and generates $\mathbf{s}_3$. It detects that $\mathbf{s}_3$ and $\mathbf{s}_1$ have the same location, and the parametric zone in $\mathbf{s}_3 : (l_2; y > p_1)$ is larger than the zone in $\mathbf{s}_1 : (l_2; y > 2p_1)$. Consequently, the state $\mathbf{s}_1$ is removed from $\mathcal{W}$ and $\mathcal{G}$ and state $\mathbf{s}_3$ is inserted at the beginning of $\mathcal{W}$. Finally, the exploration of $\mathbf{s}_1$ is stopped and $\mathbf{s}_4$ is reached.

Example 22. Consider the inefficient phenomenon in Fig. 5.1b repeated n times as in Fig. 5.5 with parameter $p > 0$. Then it is inefficient for the ranking strategy, BFS. The performance of each algorithm with this model is given in Section 5.5.

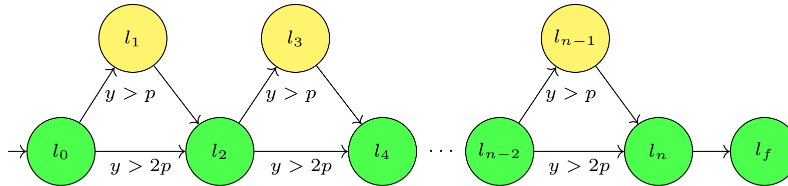


Figure 5.5 – Blowup example

However, our approaches still have some drawbacks. First, our algorithms base on the BFS so that before ranking, from a state, it has to reach all its descendants. This blind exploration might cause the inefficient phenomenon to happen, as illustrated in the previous example. Second, there might exist many paths between a pair of states that have equal parametric zones at start and different at the end, or some paths having small parametric zones in the

Algorithm 13: Parametric priority strategy algorithm

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, K_0, I, E)$
Output: parametric zone graph $\mathcal{Z}$ associated with the PTA $\mathcal{A}$

```
1  W ← {(l0, C0)}
2  G ← {(l0, C0)}
3  while W ≠ ∅ do
4      pick the first state (l, C) from W
5      foreach outgoing state (l', C') from (l, C) do
6          if there is no (l', C_Larger) ∈ G such that C' ⊆ C_Larger then
7              add (l', C') to G
8              add transition (l, C) → (l', C') to G
9              foreach (l', C_Smaller) ∈ G such that C_Smaller ⊆ C' do
10                 add transitions to G:
11                 parent states of (l', C_Smaller) → (l', C') and
12                                   (l', C') → children states of
                    (l', C_Smaller)
13                 remove (l', C_Smaller) from W and G
14             if C' = true then insert (l', C') before the head of W
15             else if C' ≠ true then
16                 insert (l', C') before the first state (l_S, C_S) with a smaller
17                 zone C_S ⊆ C' found in W
18             else
19                 insert (l', C') at the end of W
20         else
21             foreach (l', C_Larger) ∈ G such that C' ⊆ C_Larger do add
                transition (l, C) → (l', C_Larger) to G
22 return G
```

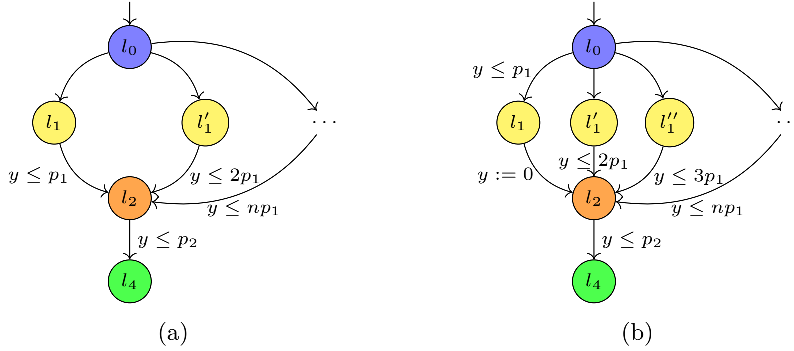


Figure 5.6 – Inefficiency in largest zone first like algorithms

beginning of the path that become larger after taking reset transitions as in Fig. 5.6a and Fig. 5.6b respectively. In Fig. 5.6a, the parametric zones of l_1, l_1', l_1'' $\dots l_1^n$ are equal. Thus, the algorithms explore from l_1 to l_1^n, but at the first state l_1, the parametric zone of l_2 reached has the smallest parametric zone. Hence, it causes the inefficient phenomenon repeatedly. In Fig. 5.6b, after all descendants of l_0 are reached, the parametric zone in l_1 is the smallest, then the path from l_1 is explored last. But it should be explored first, because the parametric zone at l_2 is reached from l_1 and is the biggest one due to resetting the clock y on the transition. Thus it causes the inefficient phenomenon.

5.5 Experimental evaluation

To evaluate the performances of the proposed exploration orders experimentally, we compare them with one another, as well as with the standard BFS exploration strategy.

We implemented our algorithms in IMITATOR [AFKS12][1], and ran our experiments on an Intel core 2 duo P8600 processor at 2.4 GHz with 4 GiB of RAM. The machine was running Ubuntu 16.04 LTS 64-bit and the code was compiled using OCaml 4.02.3. Polyhedra operations are performed using the PPL 1.2 library [BHZ08].

Our benchmarks come from the IMITATOR benchmarks library and include hardware circuits (`AndOr`, `flipflop`, `spsmall`), network or software protocols (`BRP`, `FDDI-2`, `FDDI-4`, `Fischer-2`, `Fischer-3`, `F3`, `F4`, `Lynch-2`, `Lynch-5`, `critical-region`, `RCP`), real-time systems (`Thales-1`, `Thales-3`, `Sched2.`$i.j$), variants of a producer-consumer (`Pipeline` [KP12]), and the additional `blowup` example from Fig. 5.5 with 1001 locations.

We mainly focus on reachability synthesis, recall the EF-synthesis problem: “find all parameter valuations for which a given location is reachable”. A semi-

[1] Working version 2.9.2 (build 2363 – `explorder/5c40e39`). Sources, binaries, models, logs are available at `www.imitator.fr/static/ICECCS17/`.

algorithm was proposed in [AHV93, JLR15], which we call EFsynth.

Additionally, we also focus on the counter-example synthesis: "find at least some parameter valuations for which a given location is reachable". Counter-example synthesis is of high practical importance, as it is often desirable to find at least some valuations for which a property holds (or is violated), not necessarily all of them. We implemented a procedure EFc-ex that stops as soon as some valuations are synthesized. Due to the undecidability of the EF-emptiness problem, neither EFsynth nor EFc-ex are guaranteed to terminate; note that they do for most of our experiments but not always. An advantage is that EFc-ex has a better termination than EFsynth (and in fact terminates in all our case studies) as a smaller part of the state space needs to be explored.

We will compare our new exploration strategies, i. e. parametric ranking strategy (RS) and parametric priority strategy (PRIOR), with the classical breadth-first search (BFS) strategy. In addition, we also consider a layer-based BFS strategy (LayerBFS), which is the historical strategy in IMITATOR, that computes all successor states of a given depth before computing all their successors at once. Although this is very close to the classical BFS strategy, some subtle implementation differences make its performances slightly different from BFS and significantly when the merging heuristics (see Section 5.5.1 below) is used. Both BFS and LayerBFS come with two flavors: the bidirectional inclusion incl2 (which is as in Algorithm 9) and the mono-directional inclusion incl, where we only test whether the new state is included into an existing state, but not the other way round (i. e. lines 9–12 are discarded).

5.5.1 Symbolic state merging

Our comparison is not entirely fair, as we did not use in our experiments another efficient optimization implemented in IMITATOR, i. e. state merging [AFS13a]. Given two states $\mathbf{s}_1 = (l_1, C_1)$ and $\mathbf{s}_2 = (l_2, C_2)$, it is possible to try to *merge* these states: $\mathbf{s}_1$ and $\mathbf{s}_2$ are *mergeable* if $l_1 = l_2$ and the polyhedron $C_1 \cup C_2$ is convex. The *merging* of $\mathbf{s}_1$ and $\mathbf{s}_2$ is then $(l_1, C_1 \cup C_2)$. In [AFS13a], the authors showed that merging states while computing the symbolic states keeps the soundness of the EFsynth algorithm; however, other algorithms usually lose their soundness when using state merging (this is the case of trace preservation synthesis, also called *inverse method* [AFS13a]). For example, parametric deadlock freeness checking [And16] resembles EFsynth, but merging was not proved to be sound. Despite the very high cost of the mergeability test (up to 1000 times slower than other operations on polyehdra), state merging is often efficient because it can dramatically reduce the state space, especially for the verification of parametric schedulability problems.

We compare in Tables 5.2a and 5.2b our exploration strategies with state merging. This time, our strategies are less efficient for exact synthesis using EFsynth (Table 5.2a); overall, the historical exploration strategy LayerBFS implemented in IMITATOR behaves about 2 times better than all other strategies. A reason comes from the cost of the merging: testing mergeability is very expensive, and LayerBFS iteratively tries to merge states once every state space

depth ("layer") is completed, while other strategies try to merge states *for each newly computed symbolic state*, which is much more expensive. However, even when merging is used, our new strategies RS and PRIOR preserve a dramatic decrease of the computation time of more than 75 % for EFc-ex (Table 5.2b).

5.5.2 Comparison

We compare in Tables 5.1a and 5.1b our exploration strategies. From left to right in each table are the model's name followed by the computation times in seconds for each of the four strategies. Note that the green and yellow cells are the fastest and the second-fastest approaches respectively, and "TO" stands for time-out after 15 minutes.

In order to compare all algorithms, in the last line, we compute an average of the *normalized* computation times. However, due to the variety of the computation times (a same algorithm can use 0.014 s for a benchmark and 628 s for another one), performing the actual average wouldn't be fair: the behavior of the algorithms for the slowest benchmarks would have a much larger impact on the average than the fast benchmarks.

As a consequence, we normalize all computation times as follows: for each benchmark, we replace each computation time t with the division of this computation time t by the fastest algorithm for this benchmark i. e.

$$\textit{normalized} = \frac{t}{\min_{algorithm} t_{algorithm}}$$

That is, the fastest algorithm becomes 1 (which is the smallest possible value), and other timings give an idea of how slow they are w.r.t. the fastest. For example, a normalized value of 5.4 denotes that the algorithm is 5.4 times slower than the fastest algorithm for this benchmark.

In addition, to avoid that an algorithm gets a huge penalty for being, e. g. 200 times slower for one case study, we cap this normalized timing by 10. That is, the final formula becomes:

$$\textit{normalized} = \min\big(\frac{t}{\min_{algorithm} t_{algorithm}}, 10\big)$$

Similarly, a timeout becomes 10 as well. Finally, the average given in the tables is the average of all normalized times of an algorithm.

From Table 5.1a, our two strategies RS and PRIOR behave almost the same for EFsynth, with a normalized average of 2.8. They both improve BFS by about 20 %, which shows the efficiency of our strategies.

From Table 5.1b, our strategies RS and PRIOR behave again almost the same for EFc-ex, but improve this time dramatically the computation time w.r.t. BFS, with a decrease of about 80 %. This shows the high efficiency of our strategies for counter-example synthesis.

A reason for the much better efficiency of our strategies for EFc-ex than EFsynth is that our strategies try to explore the largest zones first, and

intuitively may lead much faster to a goal state. Then, once a goal state is found, EFc-ex stops and returns the associated parameter valuations, whereas EFsynth must explore the rest of the state space, for which the benefit of our strategies is milder.

5.5.3 Final interpretation

Let us summarize the outcomes of our experiments.

Exact synthesis When one is interested in the exact synthesis (i. e. find all parameter valuations using EFsynth) for only reachability properties, then merging can be used, and the results are presented in Table 5.2a: the fastest strategy is clearly LayerBFS. Mono or bi-directional state inclusion do not fundamentally change the computation times, but in most cases (and in average), the bi-directional state inclusion is most efficient.

When one is interested in the exact synthesis for non-necessarily reachability properties, then merging cannot be used, and the results are tabulated in Table 5.1a: our two strategies RS and PRIOR perform best, 20 % faster than existing strategies.

Partial synthesis When one is interested in finding some valuations only (i. e. EFc-ex), our two strategies RS and PRIOR perform significantly better than existing strategies, with a division of the computation time by 5 in average, when comparing with existing strategies.

As most computation times are below 1 s (and often below 0.1 s), the differences between RS and PRIOR, or with or without merging, are not significant. Only Lynch-5 may suggest to use PRIOR without merging (Table 5.1b); the normalized average time (for PRIOR) also suggests this.

Overall, PRIOR is 5.7 times faster than LayerBFS and 5.0 times faster than BFS using the normalized averages; for some case studies, the improvement w.r.t. BFS grows to 33 (`blowup`), 41 (`spsmall`), 72 (`Thales-3`), or even 522 (`pipeline-KP12-3-3`). Also note that, with the exception of `blowup`, the aforementioned three case studies are all industrial case studies.

5.6 Conclusion

In this chapter, we have proposed two exploration order strategies to mitigate the inefficient phenomenon for the parameter synthesis problem: the parametric ranking strategy, and the parametric priority strategy. The intuition behind our strategies is to explore the large parametric zone first before reaching smaller zones. And to the best of our knowledge, the inefficient phenomenon is inevitable for PTA and TA.

Overall, our new strategies are reasonably faster than existing approaches for EFsynth, except when the merging heuristics is used (in which case BFS is more efficient). Our strategies become much faster than in the literature for

EFsynth (without merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	2.512	2.386	2.41	2.38	1.708	1.714
flipflop	121.108	121.026	102.42	109.412	139.822	140.193
BRP	377.913	322.67	370.74	304.277	174.038	160.079
Thales-1	30.802	37.75	44.114	37.593	41.575	40.476
Thales-3	627.956	TO	759.987	TO	636.823	597.57
Sched2.100.0	2.066	2.185	1.924	1.976	1.886	1.899
Sched2.100.2	148.169	93.138	TO	90.158	249.373	259.895
Sched2.50.0	1.649	1.779	1.6	1.602	1.57	1.607
Sched2.50.2	28.137	27.119	217.399	23.344	36.81	35.26
FDDI-2	0.014	0.018	0.009	0.009	0.009	0.01
FDDI-4	1.315	1.252	1.1	1.092	1.455	1.285
Fischer-2	0.052	0.048	0.041	0.04	0.04	0.041
Fischer-3	0.521	0.538	0.48	0.497	1.172	1.316
Lynch-2	0.047	0.04	0.03	0.029	0.03	0.027
Lynch-5	7.359	7.429	7.817	7.346	8.859	7.867
F3	0.289	0.285	0.289	0.288	0.093	0.088
F4	21.813	22.573	37.558	20.626	108.629	96.983
Pipeline-KP12-2-3	21.975	31.642	18.516	29.735	19.489	19.07
Pipeline-KP12-2-5	TO	TO	TO	TO	TO	TO
Pipeline-KP12-3-3	TO	TO	TO	TO	TO	TO
RCP	1.105	1.147	1.099	1.088	0.093	0.095
spsmall	10.132	10.99	9.595	9.883	11.114	10.232
critical-region	TO	TO	TO	TO	TO	TO
critical-region-4	TO	TO	TO	TO	TO	TO
blowup	31.635	31.758	1.345	1.32	1.493	1.134
Average	3.47236	3.82884	3.7417	3.36366	2.85594	2.81208

(a) EFsynth (without merging)

EFsynth (with merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	1.635	1.618	1.734	1.758	5.365	5.305
flipflop	91.89	87.626	80.264	83.156	243.631	244.531
BRP	304.798	313.73	TO	TO	501.384	515.021
Thales-1	10.479	10.62	63.062	66.343	60.893	65.504
Thales-3	130.026	104.372	TO	TO	TO	TO
Sched2.100.0	2.55	2.771	7.996	8.302	17.352	17.327
Sched2.100.2	131.457	98.716	456.22	375.883	TO	TO
Sched2.50.0	2.078	2.213	6.185	6.405	14.45	14.231
Sched2.50.2	41.648	35.196	121.428	110.747	224.544	229.708
FDDI-2	0.015	0.01	0.009	0.008	0.011	0.008
FDDI-4	1.476	1.225	1.316	1.307	2.063	2.01
Fischer-2	0.048	0.046	0.046	0.039	0.042	0.042
Fischer-3	0.418	0.419	0.454	0.466	2.058	2.082
Lynch-2	0.031	0.036	0.028	0.026	0.044	0.037
Lynch-5	3.852	4.101	12.928	13.457	14.901	14.339
F3	0.244	0.245	0.409	0.412	0.143	0.137
F4	19.219	18.197	TO	TO	622.301	546.115
Pipeline-KP12-2-3	0.586	0.589	0.557	0.565	30.234	75.238
Pipeline-KP12-2-5	2.796	2.793	7.31	7.348	TO	TO
Pipeline-KP12-3-3	96.509	87.764	TO	TO	TO	TO
RCP	1.094	1.144	7.805	7.326	0.086	0.09
spsmall	1.893	1.591	2.623	2.547	12.498	13.677
critical-region	TO	TO	TO	TO	TO	TO
critical-region-4	TO	TO	TO	TO	TO	TO
blowup	313.649	33.018	345.481	1.365	1.709	1.343
Average	2.58396	2.52052	4.77691	4.38764	5.58949	5.59692

(b) EFc-ex (with merging)

Table 5.1 – Experiments

EFc-ex (without merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	0.012	0.009	0.011	0.009	0.008	0.008
flipflop	0.061	0.064	0.059	0.055	0.029	0.028
BRP	2.874	2.476	2.944	2.863	0.198	0.188
Thales-1	4.854	4.753	6.189	5.748	0.126	0.119
Thales-3	16.638	16.397	19.968	20.247	0.237	0.232
Sched2.100.0	0.018	0.007	0.01	0.004	0.004	0.005
Sched2.100.2	0.008	0.007	0.004	0.004	0.005	0.004
Sched2.50.0	0.028	0.031	0.025	0.02	0.022	0.016
Sched2.50.2	0.028	0.036	0.023	0.024	0.016	0.015
FDDI-2	0.008	0.008	0.005	0.01	0.005	0.008
FDDI-4	0.377	0.329	0.291	0.287	0.091	0.078
Fischer-2	0.03	0.026	0.025	0.022	0.02	0.016
Fischer-3	0.097	0.097	0.097	0.092	0.057	0.059
Lynch-2	0.033	0.034	0.034	0.031	0.032	0.029
Lynch-5	7.408	7.619	7.912	7.329	8.847	7.829
F3	0.249	0.245	0.252	0.253	0.059	0.055
F4	4.086	3.786	6.543	4.16	0.364	0.311
Pipeline-KP12-2-3	0.313	0.344	0.245	0.27	0.031	0.025
Pipeline-KP12-2-5	3.825	4.83	2.988	4.526	0.049	0.037
Pipeline-KP12-3-3	21.927	33.184	18.229	31.204	0.042	0.042
RCP	0.51	0.506	0.454	0.453	0.024	0.02
spsmall	5.862	6.207	6.242	5.989	0.143	0.143
critical-region	0.148	0.121	0.081	0.073	0.016	0.018
critical-region-4	1.008	0.95	0.821	0.844	0.044	0.043
blowup	32.893	32.828	1.346	1.35	1.337	1.003
Average	6.03173	5.84164	5.28516	5.20335	1.13675	1.06026

(a) EFSYNTH (without merging)

EFc-ex (with merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	0.011	0.011	0.009	0.009	0.012	0.008
flipflop	0.074	0.065	0.061	0.060	0.031	0.028
BRP	1.906	2.215	3.950	4.112	0.198	0.197
Thales-1	3.191	3.182	8.461	9.068	0.126	0.123
Thales-3	10.826	11.233	55.746	58.321	0.244	0.234
Sched2.100.0	0.006	0.007	0.008	0.006	0.005	0.005
Sched2.100.2	0.009	0.01	0.006	0.006	0.004	0.01
Sched2.50.0	0.026	0.032	0.023	0.023	0.016	0.016
Sched2.50.2	0.036	0.037	0.024	0.024	0.017	0.017
FDDI-2	0.013	0.007	0.007	0.007	0.011	0.006
FDDI-4	0.364	0.365	0.35	0.349	0.088	0.08
Fischer-2	0.028	0.024	0.021	0.022	0.016	0.017
Fischer-3	0.085	0.093	0.084	0.09	0.072	0.069
Lynch-2	0.034	0.03	0.033	0.029	0.032	0.034
Lynch-5	3.893	4.182	13.038	13.533	14.778	14.344
F3	0.199	0.209	0.34	0.333	0.131	0.12
F4	3.048	3.151	45.059	53.607	0.948	0.867
Pipeline-KP12-2-3	0.096	0.086	0.075	0.082	0.029	0.028
Pipeline-KP12-2-5	0.278	0.267	0.247	0.273	0.038	0.04
Pipeline-KP12-3-3	0.639	0.639	0.825	0.841	0.043	0.041
RCP	0.532	0.575	3.117	3.154	0.022	0.024
spsmall	1.227	1.119	1.752	1.835	0.266	0.14
critical-region	0.111	0.112	0.082	0.072	0.025	0.015
critical-region-4	1.092	0.78	1.109	1.212	0.043	0.044
blowup	320.011	32.81	348.86	1.356	1.417	1.093
Average	5.05095	4.98319	5.21204	4.86171	1.29771	1.20662

(b) EFc-ex (with merging)

Table 5.2 – Experiments

the counter-example synthesis using EFc-ex, up to 522 times faster for some industrial case studies. This suggests to use our new strategies as default for counter-example synthesis.

Furthermore, in order to verify liveness properties efficiently, the parametric priority strategy is then reused in the next chapter for our new Depth-First Search based algorithms. We will show how this strategy can help our algorithms to avoid infinite runs and return complete results.

CHAPTER

6

Layered and Collecting NDFS with Subsumption for Parametric Timed Automata

Contents

6.1 Introduction **104**

6.2 Preserving accepting runs with subsumption **106**

6.3 Parametric timed nested depth-first search with subsumption **107**

6.3.1 NDFS with subsumption for PTA 107

6.3.2 Early pruning of the red search 110

6.3.3 Starting the red search early: A layered NDFS . . . 110

6.4 Collecting NDFS for parameter synthesis **112**

6.5 Experiments **112**

6.5.1 Implementation 114

6.5.2 Experimental results 114

6.6 Conclusion **117**

6.1 Introduction

In Chapter 5, several strategies were studied to exhibit efficient exploration orders for Breadth-First Search (BFS), that address parameter synthesis for

reachability properties.

However, reachability properties are often insufficient, and it is necessary to model-check liveness properties. These are generally described as a Linear Temporal Logic (LTL) formula. The LTL model-checking approach boils down to a Büchi emptiness problem. This is achieved by the search of accepting cycles in the state space, typically in a Depth-First Search fashion.

Contribution This work presents DFS exploration to find accepting cycles for Parametric Time Automata, using several reductions of the state space and smart exploration orders.

1. First, the subsumption of [LOD+13] for Timed Automata, that reduces the explored state space while preserving cycle detection, is extended to the parametric case.
2. Second, PTAs enjoy properties of the projection of zones on the parameters which can be used to stop the exploration of a branch at an early stage or check for cycles in layers of the graph. Note that such a layered approach could also be used for other types of models which exhibit some progress measure, as is done with the sweep-line exploration [EK14].
3. Finally, DFS for LTL model-checking usually exits as soon as an accepting cycle is found. However, in the case of parameter synthesis, it is desirable to obtain all parameters for which such a cycle exists. As they may lie in different branches, the exploration must be continued and the parameters zones collected all along the process. This results in a collecting algorithm.

Related work LTL parameter synthesis for PTA was addressed in [BBBC16], where parameter valuations are restricted to bounded integers. We make no such restriction, giving up decidability of the problem. Nevertheless, our semi-algorithm either gives an exact result or does not terminate. Although extrapolation is used in the model checking algorithm of [BBBC16] as a form of abstraction, there was no early pruning, subsumption, or layered verification. Therefore branches had to be explored in depth, which was feasible only due to the fact that the system is finite after extrapolation. The pruning and layering introduced in our algorithm sometimes avoids infinite branches, and provides speedup for the finite case, as demonstrated in the experiments.

Our basic algorithm extends NDFS with subsumption from [LOD+13] from the setting of TA to Parametric TA. Due to the nature of the parametric zones we identified additional pruning opportunities. Another extension of NDFS with subsumption for TA was studied in [HSTW16]. These authors proved that LTL model checking for TA is inherently harder than reachability checking for TA. They also proposed to search accepting cycles in the subsumed state space first. If there are no such cycles, the original system is correct. Otherwise, the Strongly Connected Components that contain accepting cycles must be further refined, since subsumption may introduce accepting cycles. Our layered approach

also tends to find abstract cycles first, but it is fully integrated in the NDFS procedure.

Outline We first give preserving accepting runs with subsumption for PTBA in Section 6.2. Then we provide an extension of the *ndfs* algorithm with subsumption for PTBA in Section 6.3. The parameter synthesis requires obtaining of all possible contraints on accepting cycles, as described by the collecting algorithm in Section 6.4 together with some pruning optimisations. The experimental evaluation of these algorithms is detailed in Section 6.5. Finally, Section 6.6 concludes and provides insight on future work.

6.2 Preserving accepting runs with subsumption

Recall that an accepting run from $\mathbf{s}$ is an infinite run $\mathbf{s} = \mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \cdots$ that hits an accepting location infinitely often.

Proposition 6.2.1. *If there exists an accepting run from $\mathbf{s}$ and $\mathbf{s} \sqsubseteq \mathbf{s}'$, then there exists an accepting run from $\mathbf{s}'$.*

Proof. By Proposition 2.5.1 an infinite run from $\mathbf{s}$, $\mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \cdots$, can be simulated by an abstract infinite run from $\mathbf{s}'$, $\mathbf{s}'_1 \Rightarrow \mathbf{s}'_2 \Rightarrow \cdots$, where each $\mathbf{s}_i \sqsubseteq \mathbf{s}'_i$. By Definition 2.5.5, the new run passes through the exact same locations, so it is still accepting. □

Proposition 6.2.1 shows that a subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, $\mathsf{PZG}_\alpha(\mathcal{B})$ may introduce accepting runs which were not present in $\mathsf{PZG}(\mathcal{B})$. This is already the case for (plain) timed automata [LOD+13]. This can be illustrated by their example which is prensented in Fig. 6.1. The figure visualises $\mathsf{PZG}_\alpha(\mathcal{B})$ by drawing subsumed states inside subsuming states (e. g. $s_3 \sqsubseteq s_1$).

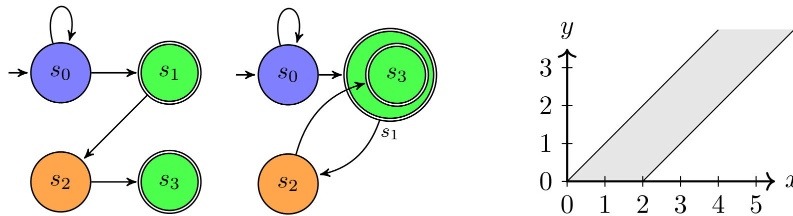


Figure 6.1 – The symbolic state space $\mathsf{PZG}(\mathcal{B})$ of the model in Fig. 2.7b without information on contraints

Taken from [LOD+13]. Consider the TA obtained from the PBTA in Fig. 2.7b, by setting $p = q = r = 2$. The symbolic state space $\mathsf{PZG}(\mathcal{B})$ of this TA, with $\ell_1 \in F$, contains 4 states (shown on the left): $\mathbf{s}_0$, $\mathbf{s}_1 = (\ell_1, C_1)$, $\mathbf{s}_2 = (\ell_2, C_2)$ and $\mathbf{s}_3 = (\ell_1, C_3)$. Note that there is no accepting cycle. The graphical representation of the zones C_1–C_3 (right) reveals that $C_3 \subseteq C_1$ and hence $\mathbf{s}_3 \sqsubseteq \mathbf{s}_1$. As $\mathbf{s}_3 \sqsubseteq \mathbf{s}_1$

and both are reachable, the subsumption abstraction might map $\alpha(\mathbf{s}_3) = \mathbf{s}_1$, introducing a cycle $\mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \mathbf{s}_1$ in $\mathsf{PZG}_\alpha(\mathcal{B})$.

Still, subsumption implies a property on paths that we can use. We adapt the results from [LOD+13] to a setting where PZG might be infinite. In subsequent sections, we exploit these properties to improve algorithms that implement the PTBA emptiness check.

Proposition 6.2.2. *If* $\mathbf{s} \Rightarrow^* \mathbf{s}'$, $\mathbf{s} \sqsubseteq \mathbf{s}'$ *and* $\mathbf{s} \in F$, *then* $\mathsf{PZG}(\mathcal{B})$ *has an accepting run.*

Proof. Let $\mathbf{s} \sqsubseteq \mathbf{s}'$ and $\mathbf{s} \in F$, then by Definition 2.5.5, $\mathbf{s}' \in F$. Let $\mathbf{s} \Rightarrow^* \mathbf{s}'$, then by Proposition 2.5.1 we can simulate this same trace from $\mathbf{s}'$, leading to some $\mathbf{s}''$ s.t. $\mathbf{s}' \Rightarrow^* \mathbf{s}''$ and $\mathbf{s}' \sqsubseteq \mathbf{s}''$ and $\mathbf{s}'' \in F$. This process can be repeated to obtain an infinite accepting run. □

We finish this section with an observation for the special case of a finite abstraction $\mathsf{PZG}_\alpha(\mathcal{B})$:

Proposition 6.2.3. *If* $\mathsf{PZG}_\alpha(\mathcal{B})$ *is finite and does not contain an accepting cycle, then* $\mathsf{PZG}(\mathcal{B})$ *does not contain an accepting run, hence* $\mathcal{L}(\mathcal{B}) = \emptyset$.

Proof. Let $\mathsf{PZG}(\mathcal{B})$ contain an accepting run from $\mathbf{s}_0$. Then $\mathsf{PZG}_\alpha(\mathcal{B})$ contains an accepting run as well from its initial state $\alpha(\mathbf{s}_0)$, by Proposition 6.2.1. Since $\mathsf{PZG}_\alpha(\mathcal{B})$ is finite and the accepting run visits infinitely many accepting states, some accepting state, say $\mathbf{s} \in F$ is visited infinitely often. Then we can construct an accepting cycle $\alpha(\mathbf{s}_0) \Rightarrow^* \mathbf{s} \Rightarrow^* \mathbf{s}$. □

6.3 Parametric timed nested depth-first search with subsumption

This section extends the NDFS algorithm with subsumption for TA [LOD+13] to PTA and introduces early checks and cuts to optimise the search. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for finite graphs.

6.3.1 NDFS with subsumption for PTA

In the following, with *soundness*, we mean that when NDFS reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that NDFS always reports an accepting cycle if the graph contains one.

The classical NDFS algorithm shown in Algorithm 14 consists of an outer DFS (*dfsBlue*) that sorts accepting states $\mathbf{s}$ in DFS *post-order* and an inner DFS (*dfsRed*) that searches for cycles over each $\mathbf{s}$, called the *seed*. States are maintained in 3 colour sets:

1. *Blue*, states *explored* by *dfsBlue*,

Algorithm 14: Classical NDFS

```
1  procedure ndfs
2      foreach  Cyan := Blue := Red := ∅ do
3          dfsBlue(s_0)
4          report no cycle
5  procedure dfsBlue(s)
6      Cyan := Cyan ∪ {s}
7      foreach t in NEXT-STATE(s) do
8          if t ∉ Blue ∧ t ∉ Cyan then
9              dfsBlue(t)
10     if s ∈ F then
11         dfsRed(s)
12     Blue := Blue ∪ {s}
13     Cyan := Cyan \ {s}
14 procedure dfsRed(s)
15     Red := Red ∪ {s}
16     foreach  t in NEXT-STATE(s) do
17         if t ∈ Cyan then
18             report cycle
19         if t ∉ Red then
20             dfsRed(t)
```

2. *Cyan*, states on the stack of *dfsBlue* (*visited* but not yet explored), and

3. *Red*, visited by *dfsRed*.

The goal of the blue search is to visit all states, and launch a red search on all accepting states in post-order. The goal of the red search is to detect cycles on these accepting states, by finding edges to cyan states. These would close cycles early, at line 17 [SE05].

The NDFS-algorithm with subsumption from [LOD+13] for TA is presented in Algorithm 15. Here the parts introduced by subsumption are highlighted in light blue. We now show why the same algorithm is valid for PTA as well.

As for TA, subsumption can be used to prune both searches, but we should be careful, since $\mathsf{PZG}_{\sqsubseteq}(\mathcal{B})$ may introduce additional cycles (Fig. 6.1). To express subsumption checks on sets we write $\mathbf{s} \sqsubseteq S$, meaning $\exists \mathbf{s}' \in S \colon \mathbf{s} \sqsubseteq \mathbf{s}'$. And $S \sqsubseteq \mathbf{s}$, meaning $\exists \mathbf{s}' \in S \colon \mathbf{s}' \sqsubseteq \mathbf{s}$.

According to Proposition 6.2.2, a state that subsumes a state on the cyan stack leads to a cycle. This explains the cycle detection in line 18. According to Proposition 6.2.1, if there is a cycle from t, then there is a cycle from all t' with $t \sqsubseteq t'$. So, if $t \sqsubseteq \mathit{Red}$, it cannot be on an accepting cycle, which explains subsumption in line 20. By definition (Definition 2.5.7), $\mathsf{PZG}_{\sqsubseteq}(\mathcal{B})$ contains a

Algorithm 15: NDFS with subsumption checks and red prune of *dfsBlue* (in light-blue) and early pruning (in yellow).

```
1  procedure ndfs
2  │ Cyan := Blue := Red := ∅
3  │ dfsBlue(s0)
4  └ report no cycle
5  procedure dfsBlue(s)
6  │ Cyan := Cyan ∪ {s}
7  │ forall t in NEXT-STATE(s) do
8  │ │ if t ∉ Blue ∪ Cyan ∧ t ⋢ Red then
9  │ └ └ dfsBlue(t)
10 │ if s ∈ F then
11 │ └ dfsRed((s)
12 │ Blue := Blue ∪ {s}
13 └ Cyan := Cyan \ {s}
14 procedure dfsRed(s)
15 │ Red := Red ∪ {s}
16 │ foreach t in NEXT-STATE(s)
17 │ s.t. t =_p s do
18 │ │ if Cyan ⊑ t then
19 │ │ └ report cycle
20 │ │ if t ⋢ Red then
21 └ └ └ dfsRed(t)
```

"larger" state for all reachable states in $\mathsf{PZG}(\mathcal{B})$, so this is sufficient to find all accepting cycles.

Since red states do not lead to accepting cycles, red states can even prune the blue search. We can strengthen the condition on line 8 to $t \notin \mathit{Blue} \cup \mathit{Cyan} \cup \mathit{Red}$. However, this is by itself of no use since, $\mathit{Red} \subseteq \mathit{Blue}$. Luckily, even states subsumed by red do not lead to accepting cycles (contraposition of Proposition 6.2.1), so we can use subsumption again: $t \notin \mathit{Blue} \cup \mathit{Cyan} \wedge t \not\sqsubseteq \mathit{Red}$, as in line 8. The benefit of this can be illustrated using Fig. 6.1. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in \mathit{Red}$ by *dfsRed* at line 11. Any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Finally, if the algorithm reaches line 4, the algorithm has completely traversed the subsumed state space $\mathsf{PZG}_{\sqsubseteq}(\mathcal{B})$, which was apparently finite. Since no accepting cycle was detected, by Proposition 6.2.3 the algorithm may conclude that $\mathcal{B}$ is empty.

Algorithm 15 shows a version of NDFS with all correct improvements. The part highlighted in yellow is specific for PTA, and will be explained in the next

subsection.

6.3.2 Early pruning of the red search

Some simple observation allows for avoiding unnecessary explorations: the projection of a zone C onto the parameter set P, $C{\downarrow_P}$, decreases along a path.

Notation 1. Let $\mathbf{s} = (l, C)$ and $\mathbf{s}' = (l', C')$. By $\mathbf{s} =_p \mathbf{s}'$ we denote that $C{\downarrow_P} = C'{\downarrow_P}$, i. e. they have the same parametric zone. Similarly, by $\mathbf{s} \subseteq_p \mathbf{s}'$ we denote that $C{\downarrow_P} \subseteq C'{\downarrow_P}$.

Note that in both notations, the locations l and l' may be different, as opposed to the requirement for $\sqsubseteq$.

Proposition 6.3.1. *Let* $\mathbf{s}, \mathbf{s}'$ *be two states, s.t.* $\mathbf{s} \Rightarrow \mathbf{s}'$*. Then* $\mathbf{s}' \subseteq_p \mathbf{s}$*.*

Proof. Let $\mathbf{s} = (l, C)$ and $\mathbf{s}' = (l', C')$. By Definition 2.5.4: $(l, C) \stackrel{e}{\Rightarrow} (l', C')$ if $e = (l, g, a, R, l')$ and $C' = \big([(C \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')$.

Since the projection onto P does not contain clocks, it is not affected by resets nor time elapsing. Hence, we have:

$$\begin{array}{rcl} C'{\downarrow_P} & = & \Big[\big([(C \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')\Big]{\downarrow_P} \\ & = & \Big[\big((C \wedge g){\downarrow_P} \wedge I(l')\big){\downarrow_P} \wedge I(l')\Big]{\downarrow_P} \\ & = & \big((C \wedge g){\downarrow_P} \wedge I(l')\big){\downarrow_P} \\ & \subseteq & \big(C{\downarrow_P} \wedge I(l')\big){\downarrow_P} \\ & \subseteq & C{\downarrow_P} \end{array}$$

□

Proposition 6.3.1 allows for an early pruning of the red search, as highlighted in yellow in Algorithm 15. Indeed, the red search aims at finding a cycle, which necessarily has the same parametric zone $C_{\mathbf{s}}{\downarrow_P}$ for all its states. Thus, if a successor has a smaller parametric zone $C_t{\downarrow_P} \subset C_s{\downarrow_P}$, it cannot be part of the cycle, so should not be considered by the red search.

6.3.3 Starting the red search early: A layered NDFS

As seen in Section 6.3.2, all symbolic states in an accepting cycle have the same parametric zone $C{\downarrow_P}$. From this property, we can organise our search by considering *layers* of states with the same parametric zone. Contrary to standard NDFS (for automata or timed automata), a red search in a layer of the PZG cannot interfere with a red search in another layer of the same path, since they concern different parametric zones.

Thus the new *layered* NDFS shown in Algorithm 16 works layer by layer, looking at the larger parametric zones first. The changes from Algorithm 15 are highlighted in yellow.

Notation 2. With $t \sqsubseteq_{=_p} X$ we denote that $\exists t' \in X . t \sqsubseteq t' \wedge t' =_p t$. That is, t is subsumed by some element of X in the same parametric layer.

Algorithm 16: Layered NDFS

```
1  procedure layered_ndfs
2      Cyan := Blue := Red := ∅
3      Pending := {s0}
4      while Pending ≠ ∅ do
5          Pick s from Pending
6          if s ∉ Blue then
7              dfsBlue(s)
8          Pending := Pending \ {s}
9      report no cycle
10 procedure dfsBlue(s)
11     Cyan := Cyan ∪ {s}
12     foreach t in NEXT-STATE(s) do
13         if t ∉ Blue ∪ Cyan ∧t ⋢=p Red then
14             if t ⊂p s then
15                 Pending := Pending ∪ {t}
16             else
17                 dfsBlue(t)
18     if s ∈ F then
19         dfsRed(s)
20     Blue := Blue ∪ {s}
21     Cyan := Cyan \ {s}
22 procedure dfsRed(s)
23     Red := Red ∪ {s}
24     foreach t in NEXT-STATE(s) do
25         s.t. t =p s
26         if Cyan ⊑ t then
27             report cycle
28         if t ⋢=p Red then
29             dfsRed(t)
```

In order to obtain a result as fast as possible, and with the largest parametric zone as possible, we run the blue search until the parametric zone changes (line 14–line 17). The last state thus constructed is kept as *Pending* (line 15), i. e. its successors are not generated yet, and we continue the NDFS algorithm on other branches if any. When backtracking, the deepest accepting state in the layer will be encountered first, thus preserving the post-order for the red search in the layer.

If an accepting cycle is found it is reported by the algorithm, otherwise the exploration continues in the current layer. When the layer is finished, the

algorithm is applied to the pending states (line 4–line 8).

In the red search, the comparison with cyan states at line 27 is still valid. Indeed, all cyan states are on the path leading to the state examined; if one of them is subsumed by the current state t, its parametric zone is smaller than or equal to the one of t, but it is also larger than or equal to it as it is on the path. Hence they have the same parametric zone, and only the subsumption on clocks

The comparison with red states at line 29, however, needs to be limited to the current layer, so as not to interfere with a previous red search on another layer. This is sufficient since any red state encountered on the same layer should have led, during its red search, to a cycle. A similar argument applies to the comparison with red states of the same layer only in the blue search at line 13.

Remark. Such a layered NDFS can also be applied to automata or TA, provided the model exhibits some progress measure that allows for determining layers. This is similar to the sweepline approach [EK14].

6.4 Collecting NDFS for parameter synthesis

This section addresses the use of NDFS to synthesize parameter values that lead to an accepting cycle, i. e. to find all possible valuations of the parameters such that there exists an accepting run.

To achieve synthesis, finding one accepting cycle is not sufficient anymore: they should all be found. Therefore, the NDFS algorithm of Section 6.3 is extended to a *collecting* NDFS that continues the exploration.

The reporting of a cycle in Algorithm 17 does no longer exit: it just collects in *Constraints* (line 28) the constraint $C_t{\downarrow_P}$ that was just found. Moreover, in order to avoid exploring smaller parametric zones, we will only process states that are not contained in *Constraints* (line 12).

6.5 Experiments

To evaluate the performances of our algorithms, we ran our experiments on a Dell Precision 3620 i7-7700 3.60 GHz with 64 GiB memory running Linux Mint 19 beta 64 bits.

Algorithm 17: Layered collecting NDFS

1. **procedure** *layered_ndfs*
2. $Cyan := Blue := Red := \emptyset$
3. $Constraints := \emptyset$
4. $Pending := \{\mathbf{s}_0\}$
5.   **while** $Pending \neq \emptyset$ **do**
6.   Pick $\mathbf{s}$ from $Pending$
7.   **if** $\mathbf{s} \notin Blue$ **then**
8. $dfsBlue(\mathbf{s})$
9. $Pending := Pending \setminus \{\mathbf{s}\}$
10.   **return** $Constraints$
11. **procedure** $dfsBlue(\mathbf{s} = (l_{\mathbf{s}}, C_{\mathbf{s}}))$
12.   **if** $\mathbf{s}{\downarrow_P} \not\sqsubseteq Constraints$ **then**
13. $Cyan := Cyan \cup \{\mathbf{s}\}$
14.   **foreach** t ***in*** NEXT-STATE($\mathbf{s}$) **do**
15.   **if** $t \notin Blue \cup Cyan \ \wedge t \not\sqsubseteq_{=_p} Red$ **then**
16.   **if** $t \subset_p \mathbf{s}$ **then**
17. $Pending := Pending \cup \{t\}$
18.   **else**
19. $dfsBlue(t)$
20.   **if** $\mathbf{s} \in F$ **then**
21. $dfsRed(\mathbf{s})$
22. $Blue := Blue \cup \{\mathbf{s}\}$
23. $Cyan := Cyan \setminus \{\mathbf{s}\}$
24. **procedure** $dfsRed(\mathbf{s})$
25.   **foreach** $t = (l_t, C_t)$ ***in*** NEXT-STATE($\mathbf{s}$)
26.   *s.t.* $t =_p \mathbf{s}$ **do**
27.   **if** $Cyan \sqsubseteq t$ **then**
28. $Constraints := Constraints \cup C_t{\downarrow_P}$
29.   **else if** $t \not\sqsubseteq_{=_p} Red$ **then**
30. $dfsRed(t)$

6.5.1 Implementation

We implemented our algorithms in IMITATOR [AFKS12][1] with Ocaml 4.02.3 and PPL 1.2 library [BHZ08]. For better performance, in Algorithm 16 and Algorithm 17, we reuse the idea from the PRIOR strategy in Chapter 5 for implementing the *Pending* list: each explored state is inserted in a decreasing zone fashion into *Pending*, and thus the state having largest zone or the state at the beginning of *Pending* is popped out first. Besides, we use an additional index, storing information on the sets of comparable zones, to speed up the state insertion.

6.5.2 Experimental results

In our experiments, we used 26 benchmarks from the IMITATOR benchmarks library that include hardware circuits (`flipflop`, `spsmall`), network or software protocols (`BRP`, `FDDI-4`, `Fischer`, `F3`, `F4`, `Lynch-2`, `Lynch-5`, `critical-region`, `critical-region-4`, `RCP`, `simop`, `WFAS`), real-time systems (`Thales-1`, `Thales-3`, `Sched2.`$i.j$), variants of a producer-consumer (`Pipeline` [KP12]), and few additional case studies `coffee`, `train-gate`, `JLR13`.

In the experiments, we mainly focused on two parameter synthesis problems. The first problem, called ECC-EX, is the counter-example synthesis: "find at least some parameter valuations for which an accepting cycle is found". Counter-example synthesis is of high practical importance, as it is often desirable to find at least some valuations for which an accepting cycle is found, not necessarily all existing ones. We implemented a procedure ECC-EX that stops as soon as some parameter valuations allowing for reaching an accepting cycle are synthesized.

Instead of finding some parameter valuations for a single accepting cycle as in ECC-EX, the second problem addressed, ECCYCLES, synthesizes all possible valuations: "find all parameter valuations for which an accepting cycle exists".

Note that, due to the undecidability of the EF-emptiness problem [AHV93], algorithms for ECCYCLES or ECC-EX are not guaranteed to terminate.

In order to show the performance of our algorithm LAYERCOLLECTNDFSSUB (Algorithm 17) is compared with the breadth first search STATESPACE synthesis with the inclusion reduction [ANP17] which explores all possible states of the system. Evidently, it is not entirely fair that ECCYCLES only explores a part of the whole state space while STATESPACE generates it all w.r.t. the inclusion. Nevertheless, it makes sense since the reduction criterion is similar, and STATESPACE analyses reachability of accepting locations.

From left to right in Table 6.1 are the models' names followed by some information on each model (number of clocks, parameters, and locations) and computation times in seconds for each of the five algorithms. Additionally, the number of different zones of accepting cycles by found LAYERCOLLECTNDFSSUB is indicated next to it. Note that the green and yellow cells are the fastest and the second-fastest approaches respectively, for each of the two categories of algorithms, and "TO" stands for a time-out after 30 minutes.

[1] Working version 2.9.2 (Build 2389 – NestedDFS/543ca62).

				EC Algorithms			Statespace Algorithms		
Benchmark	**#**	**#**	**#**	ECc-ex			ECcycles		Statespace
Models	***X***	***P***	***L***	**NDFS (s)**	**NDFSsub (s)**	**LayerNDFSsub (s)**	**LayerCollectNDFSsub (s)**	**#Zones**	**BFS PRIORincl (s)**
BRP	7	2	22	0.231	0.237	0.035	0.043	4	176.535
coffee	2	3	4	**TO**	**TO**	0.008	**TO**	2	0.006
critical-region	2	2	20	**TO**	**TO**	**TO**	**TO**	0	**TO**
F3	3	0	18	0.026	0.026	0.006	0.003	1	0.255
F4	4	2	23	**TO**	**TO**	0.007	0.006	1	**TO**
FDDI4	13	2	34	0.305	0.235	1.260	7.004	136	1.319
FischerAHV93	2	4	13	0.010	0.009	0.013	0.013	1	0.040
flipflop	5	2	52	0.010	0.010	0.010	0.012	1	0.015
fmtv1A1-v2	3	3	15	0.060	0.057	46.723	68.223	29	14.040
fmtv1A3-v2	3	3	15	0.063	0.062	302.061	1129.284	67	215.020
JLR13	2	2	2	**TO**	**TO**	**TO**	**TO**	0	**TO**
lynch	2	1	18	0.007	0.007	0.010	0.010	5	0.016
lynch5	5	1	45	0.012	0.012	0.016	0.019	9	3.126
Pipeline-KP12-2-3	4	6	14	0.510	0.508	7.738	492.995	369	**TO**
Pipeline-KP12-2-5	4	6	18	0.791	0.787	**TO**	**TO**	0	**TO**
Pipeline-KP12-3-3	5	6	19	1.960	1.962	**TO**	**TO**	0	**TO**
RCP	6	5	48	0.024	0.013	0.023	0.034	7	10.095
Sched2.50.0	6	2	17	0.011	0.011	0.539	4.168	71	1.940
Sched2.50.2	6	2	17	0.011	0.011	**TO**	**TO**	0	**TO**
Sched2.100.0	6	2	17	0.008	0.008	0.417	3.769	31	2.425
Sched2.100.2	6	2	17	0.008	0.008	**TO**	**TO**	0	**TO**
simop	8	2	46	**TO**	**TO**	**TO**	**TO**	0	**TO**
spsmall	11	2	51	0.244	0.053	0.310	36.549	1026	5.650
train-gate	5	9	11	0.016	0.015	0.047	0.268	15	0.018
WFAS	4	2	10	0.023	0.021	0.056	**TO**	13	**TO**

Table 6.1 – Experimental comparison of Nested DFS algorithms

The table is divided into two parts by a grey vertical line so as to reflect the two distinct comparisons. The first part comprises NDFS (Algorithm 14) and its variants NDFSsub (Algorithm 15) and LayerNDFSsub (Algorithm 16). The other is a comparison between LayerCollectNDFSsub (Algorithm 17) and PRIOR with inclusion in Chapter 5, which is a BFS (breadth first search) based algorithm with optimised search order for parametric zone inclusion. The PRIOR with inclusion is also our most efficient BFS algorithm for reducing state space explosion and improving termination.

In the first part of the table, NDFSsub dominates other algorithms, since it is the fastest on more than half of benchmarks (17/25 cases). However, in 4 cases, LayerNDFSsub is even faster. It is interesting that this layered algorithm terminates very quickly in two cases where the non-layered versions do not terminate. Clearly, layering can prevent the algorithm to diverge. Note however, that in 4 other cases, the layering algorithm times out, while the non-layered algorithms provide an answer quickly. In these cases the zone graph is broad already in the top layers, so apparently the strict NDFS versions find an accepting cycle more efficiently.

NDFS and NDFSsub suffer from the undecidability and cannot terminate and reach the time-out in some benchmarks. Algorithm LayerNDFSsub was proposed to cope with the termination problem. We observe that this works in two cases: `coffee` and `F4`. By exploring the state having the largest zone first (especially true zone states), the layered algorithm can avoid exploring infinite paths having smaller zones in the beginning and thus they can also prune these paths later. The timeout in 4 other cases is not caused by infinite behaviour, but by exploring many branches in the top-layers. Here a result would have been produced with larger timeout value.

Let us interpret the second part of the table, where we compare the termination of LayerCollectNDFSsub with PRIOR. We can see that the results of LayerCollectNDFSsub are similar to those of LayerNDFSsub: only 2 more cases hit the timeout limit. Note that, for efficiency purpose, all algorithms explore states on-the-fly so that in LayerNDFSsub, we receive a zone result when some cycle containing an accepting location is found, while LayerCollectNDFSsub would continue the search. PRIOR exhibits some complimentary unterminated cases. In the terminating benchmarks, LayerCollectNDFSsub is the fastest in 9/26 cases, where PRIOR is the fastest in 8/26 cases. The reason is probably that PRIOR uses full parametric zone inclusion, which is sound for reachability but not for liveness. We conclude that both the efficiency and the termination behaviour of these algorithms are quite complementary. However, note that these algorithms address a different problem.

Finally, we note that the number of symbolic zones collected during the exploration is widely different across the benchmarks (ranging from 1-1026). Even in two of the timed-out runs, LayerCollectNDFSsub collected 2, resp. 13, parametric zones that lead to an accepting cycle (`coffee` and `WFAS`).

6.6 Conclusion

In this chapter, we have proposed a new *nested depth-first search* (NDFS) algorithm and its variants for model-checking LTL properties of Parametric Timed Automata.

In the next chapter, we will show that Zeno phenomenon is unrealistic and should be removed from our final results. Therefore, to have more reliable results, in the next chapter, we will introduce an approach called clock upper bound PTA (CUB-PTA) approach to avoid the Zeno phenomenon in parametric timed automata.

CHAPTER 7

Parametric model checking timed automata under non-Zenoness assumption

Contents

7.1 Introduction . . . 119

7.2 Undecidability of the non-Zeno emptiness problem 120

7.3 CUB-parametric timed automata . . . 121

7.3.1 CUB timed automata . . . 121

7.3.2 Parametric clock upper bounds . . . 122

7.3.3 CUB parametric timed automata . . . 123

7.3.4 CUB PTA detection . . . 124

7.3.5 Transforming a PTA into a disjunctive CUB-PTA . 127

7.4 Zeno-free cycle synthesis in CUB-PTAs . . . 129

7.5 Distributing non-Zeno parametric model checking 137

7.5.1 Master algorithms . . . 137

7.5.2 Worker algorithm . . . 138

7.5.3 Handling the case of a network of PTAs . . . 140

7.6 Experiments . . . 141

7.6.1 Evaluation of the non-distributed version . . . 141

7.6.2 Evaluation of the distributed version . . . 144

7.7 Conclusion . . . 145

7.1 Introduction

Model checking TAs consists of checking whether there exists an accepting cycle (i. e. a cycle that visits infinitely often a given set of locations) in the automaton made of the product of the TA modeling the system with the TA representing a violation of the desired property (often the negation of a property expressed, e. g. in CTL). However, such an accepting cycle does not necessarily mean that the property is violated: indeed, a known problem of TAs is that they allow Zeno behaviors. An infinite run is non-Zeno if it takes an unbounded amount of time; otherwise it is Zeno. Zeno runs are infeasible in reality and thus must be pruned during system verification. That is, it is necessary to check whether a run is Zeno or not so as to avoid presenting Zeno runs as counterexamples. For instance, liveness properties are usually meaningless unless non-Zenoness is assumed; and safety properties cannot be trusted since Zeno runs may conceal deadlocks, etc. The problem of checking whether a timed automaton accepts at least one non-Zeno run, i. e. the emptiness checking problem, has been tackled previously (e. g. [Tri99, TYB05, BG06, GB07, HSW12, WSW+15]).

We address here the synthesis of parameter valuations for which there exists a non-Zeno cycle in a PTA; this is highly desirable when performing parametric model-checking for which the parameter valuations violating the property should not allow only Zeno-runs. At the time of writing this thesis, this is the first work on parametric model checking of timed automata with the non-Zenoness assumption. Just as for TAs, the parametric zone graph of PTAs (used in e. g. [HRSV02, ACEF09, JLR15]) cannot be used to check whether a cycle is non-Zeno. Therefore, we propose here a technique based on *clock upper bound PTAs* (CUB-PTAs), a subclass of PTAs satisfying some syntactic restriction, and originating in CUB-TAs for which the non-Zeno checking problem is most efficient [WSW+15]. In contrast to regular PTAs, we show that synthesizing valuations for CUB-PTAs such that there exists an infinite non-Zeno cycle can be done based on (a light extension of) the parametric zone graph.

Contributions We make the following technical contributions in this chapter:

1. We show that the parameter synthesis problem for PTAs with non-Zenoness assumption is undecidable.
2. We show that any PTA can be transformed into a finite list of CUB-PTAs;
3. We develop a semi-algorithm to solve the non-Zeno synthesis problem using CUB-PTAs.
4. We also develop a distributed algorithm for our non-Zeno synthesis approach.
5. Finally, we implemented all our algorithms in IMITATOR and validated using benchmarks.

Outline The chapter is organized as follows. Section 7.2 shows the undecidability of non-Zeno-Büchi emptiness. We then present the concept of CUB-PTAs (Section 7.3), and show how to transform a PTA into a list of CUB-PTAs. Zeno-free parametric model-checking of CUB-PTA is addressed in Section 7.4, and experiments reported in Section 7.6. Then in Section 7.5, we distribute our CUB-PTA approach on clusters. Finally, Section 7.7 concludes and gives perspectives for future work.

7.2 Undecidability of the non-Zeno emptiness problem

In this chapter, we aim at addressing the following two problems. They both concern the existence of an infinite non-Zeno run. The first one aims at checking whether the set of parameter valuations leading to such a run is empty, while the second synthesizes such a set of valuations.

non-Zeno emptiness problem:
INPUT: A PTA $\mathcal{A}$
PROBLEM: Is the set of parameter valuations v for which there exists a non-Zeno infinite run in $\mathcal{A}[v]$ empty?

non-Zeno synthesis problem:
INPUT: A PTA $\mathcal{A}$
PROBLEM: Synthesize the set of parameter valuations v for which there exists an infinite non-Zeno run in $\mathcal{A}[v]$.

As reachability is undecidable for PTAs [AHV93], it is unsurprising that the existence of at least a parameter valuation for which there exists a non-Zeno infinite run is undecidable too. The result holds with as few as one parameter, even bounded (typically in $[0, 1]$). Let us formalize this result below.

Theorem 7.2.1. *The non-Zeno emptiness problem is undecidable for PTAs with at least four clocks and one (bounded) parameter.*

Proof. By reduction from the halting problem of a deterministic 2-counter-machine, which is undecidable [Min67]. Let us encode a 2-counter machine (2CM) using PTAs. Several encodings were proposed in the literature. We rely here on an encoding proposed in [AM15], that requires one single (bounded) parameter p and four clocks. The proof of Theorem 7.2.1 does not require to modify the 2CM encoding of [AM15]. Basically, that encoding is such that a special location $l_{\texttt{halt}}$ in the PTA is reachable iff the 2-counter machine halts. This encoding has the specificity that the unique parameter p is used not only to denote the maximum possible value of the 2 counters (which is often the case in PTA-based encodings in the literature), but also to bound the number of operations of the 2CM that the PTA can simulate. That is, for any valuation v of p, the encoding of the 2CM will stop after $v(p)$ steps.

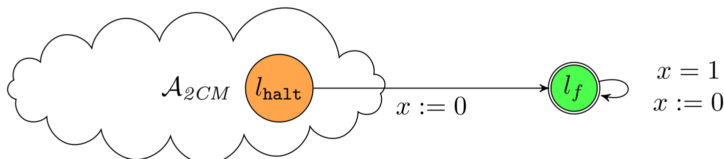


Figure 7.1 – Undecidability of the non-Zeno emptiness problem

Let $\mathcal{A}_{2CM}$ denote the PTA encoding a given 2-counter machine using the encoding of [AM15]. We extend $\mathcal{A}_{2CM}$ as shown in Fig. 7.1: from the location encoding the halting location (i. e. $l_{\texttt{halt}}$), we add a transition to a new location l_f. Then, this location has a self-loop guarded with $x = 1$ and resetting x, where x is any of the four clocks used in the encoding in [AM15].

First, recall from [AM15] that $l_{\texttt{halt}}$ is reachable iff the 2-counter-machine halts, and for parameter valuations v such that $v(p)$ is larger than or equal to the maximum value of the counters along the (unique) run of the machine. Then, once $l_{\texttt{halt}}$ is reached, l_f is reachable without condition. Then, once l_f is reached, it can be visited infinitely often every 1 time unit, thanks to the guard and the reset of x. Hence, once l_f is reached, there exists an infinite non-Zeno run for any parameter valuation for which l_f is reachable. However, if the 2CM does not halt, l_f is not reachable; furthermore, thanks to the encoding of [AM15], for any valuation, any run stops after at most $v(p)$ discrete steps, and therefore there is no infinite non-Zeno run for any valuation. Finally, any path in the encoding of [AM15] is necessarily non-Zeno as p time units elapse for each instruction of the 2-counter-machine, and p is bound to be greater than or equal to 1. Hence, there exists an infinite non-Zeno run iff the 2-counter-machine halts. □

Since the emptiness problem is undecidable, the synthesis problem becomes intractable. In the remainder of this chapter, we will devise a *semi-algorithm* to address the non-Zeno synthesis problem, i. e. an algorithm that computes the exact solution if it terminates. Otherwise, we will compute an under-approximation of the result.

7.3 CUB-parametric timed automata

7.3.1 CUB timed automata

It has been shown (e. g. [BG06, Tri99]) that checking whether a run of TA is infeasible based on the symbolic semantics alone. In [WSW+15], the authors identified a subclass of TAs called CUB-TAs for which non-Zenoness checking based on the symbolic semantics is feasible. Furthermore, they show that not only CUB-TAs are expressive enough to model most of the benchmark timed systems, but more importantly, an arbitrary TA can be transformed into a CUB-TA. Based on their work, we first show that arbitrary PTAs can be transformed

into a parametric version of CUB-TAs, and then solve the non-Zeno synthesis problem based on parametric CUB-TAs.

As defined in [WSW+15], a clock upper bound is either ∞ or a pair $(n, \lhd)$ where $n \in \mathbb{Q}$ (recall that $\lhd$ is either $<$ or $\leq$). We write $(n_1, \lhd_1) = (n_2, \lhd_2)$ to denote $n_1 = n_2$ and $\lhd_1 = \lhd_2$; $(n_1, \lhd_1) \leq (n_2, \lhd_2)$ to denote $n_1 < n_2$, or if $n_1 = n_2$, then either $\lhd_2$ is $\leq$ or both $\lhd_1$ and $\lhd_2$ are $<$. Further, we write $(n, \lhd) > d$ where d is a constant to denote $n > d$. We define $\min((n, \lhd_1), (m, \lhd_2))$ to be $(n, \lhd_1)$ if $(n, \lhd_1) \leq (m, \lhd_2)$, and $(m, \lhd_2)$ otherwise. Given a clock x and a non-parametric guard g, we write $ub(g, x)$ to denote the upper bound of x given g. Formally,

$$ub(g,x) = \begin{cases} (n, \lhd) & \text{if } g \text{ is } x \lhd n \\ \infty & \text{if } g \text{ is } x > n \text{ or } x \geq n \\ \infty & \text{if } g \text{ is } x' \bowtie n \text{ and } x' \neq x \\ \infty & \text{if } g \text{ is } \mathit{true} \\ \min(ub(g_1, x), ub(g_2, x)) & \text{if } g \text{ is } g_1 \wedge g_2 \end{cases}$$

Let us now introduce CUB-TAs.[1]

Definition 7.3.1. A TA is a *CUB-TA* if for each edge (l, g, a, R, l'), for all clocks $x \in X$, we have

1. $ub(I(l), x) \leq ub(g, x)$, and
2. if $x \notin R$, then $ub(I(l), x) \leq ub(I(l'), x)$.

Intuitively, every clock in a CUB-TA has a non-decreasing upper bound along any path until it is reset.

7.3.2 Parametric clock upper bounds

Let us define clock upper bounds in a parametric setting. A *parametric clock upper bound* is either ∞ or a pair $(\mathit{plt}, \lhd)$.

Given a clock x and a guard g, we denote by $pub(g, x)$ the parametric upper bound of x given g. This upper bound is a parametric linear term. Formally,

$$pub(g,x) = \begin{cases} (\mathit{plt}, \lhd) & \text{if } g \text{ is } x \lhd \mathit{plt} \\ \infty & \text{if } g \text{ is } x > \mathit{plt} \text{ or } x \geq \mathit{plt} \\ \infty & \text{if } g \text{ is } x' \bowtie \mathit{plt} \text{ and } x' \neq x \\ \infty & \text{if } g \text{ is } \mathit{true} \\ \min(pub(g_1, x), pub(g_2, x)) & \text{if } g \text{ is } g_1 \wedge g_2 \end{cases}$$

Recall that, in each guard, given a clock x, at most one inequality is in the form $x \lhd \mathit{plt}$. In that case, at most one of the two terms is not ∞ and therefore the minimum (last case) is well-defined (with the usual definition that $\min(\mathit{plt}, \infty) =$

[1] Note that, our definition is slightly more liberal than that in [WSW+15].

plt). Note that, if a clock has more than a single upper bound in a guard, then the minimum can be encoded as a disjunction of constraints, and our results would still apply with non-convex constraints (that can be implemented using a finite list of convex constraints).

We write $(plt_1, \lhd_1) \leq (plt_2, \lhd_2)$ to denote the constraint

$$\begin{cases} plt_1 < plt_2 & \text{if } \lhd_1 = \leq \text{ and } \lhd_2 = < \\ plt_1 \leq plt_2 & \text{otherwise.} \end{cases}$$

That is, we constrain the first parametric clock upper bound to be smaller than or equal to the second one, depending on the comparison operator.

Given two parametric clock upper bounds $pcub_1$ and $pcub_2$, we write $pcub_1 \leq pcub_2$ to denote the constraint

$$\begin{cases} (plt_1, \lhd_1) \leq (plt_2, \lhd_2) & \text{if } pcub_1 = (plt_1, \lhd_1) \text{ and } pcub_2 = (plt_2, \lhd_2) \\ \top & \text{if } pcub_2 = \infty \\ \bot & \text{otherwise.} \end{cases}$$

This yields an inequality constraining the first parametric clock upper bound to be smaller than or equal to the second one.

7.3.3 CUB parametric timed automata

We extend the definition of CUB-TAs to parameters as follows:

Definition 7.3.2. A PTA is a *CUB-PTA* if for each edge (l, g, a, R, l'), for all clocks $x \in X$, the following conditions hold:

1. $\mathcal{A}.K_0 \subseteq \big(pub(I(l), x) \leq pub(g, x)\big)$, and
2. if $x \notin R$, then $\mathcal{A}.K_0 \subseteq \big(pub(I(l), x) \leq pub(I(l'), x)\big)$.

Hence, a PTA is a *CUB-PTA* iff every clock has a non-decreasing upper bound along any path before it is reset, for all parameter valuations satisfying the initial constraint $\mathcal{A}.K_0$.

Note that, as in [WSW+15], we do not define accepting locations. In our work in this chapter, we are simply interested in computing valuations for which there is a non-Zeno cycle. A more realistic parametric model checking approach would require additionally that the cycle is accepting, it contains at least one accepting location. However, this has no specific theoretical interest, and would impact the readability of our exposé. Interestingly enough, the class of hardware circuits modeled using a bi-bounded inertial delay fits into CUB-PTAs (for all parameter valuations). This model assumes that, after the change of a signal in the input of a gate, the output changes after a delay which is modeled using a parametric closed interval.

Example 23. Consider the PTA $\mathcal{A}$ in Fig. 7.4a s.t. $\mathcal{A}.K_0 = \top$. Then $\mathcal{A}$ is not CUB: for x, the upper bound in l_0 is $x \leq 1$ whereas that of the guard on the transition outgoing l_0 is $x \leq p$. $(1, \leq) \leq (p, \leq)$ yields $1 \leq p$. Then, $\top \not\subseteq \big(1 \leq p\big)$; for example, $p = 0$ does not satisfy $1 \leq p$.

Example 24. Consider again the PTA $\mathcal{A}$ in Fig. 7.4a, this time assuming that $\mathcal{A}.K_0 = (p = 1 \wedge 1 \leq p' \wedge p' \leq p'')$. This PTA is a CUB-PTA. (The largest constraint K_0 making this PTA a CUB will be computed in Example 26.)

Example 25. The four examples of Fig. 7.2 introduce all possible cases encountered in Algorithm 18. Fig. 7.2a features a self-loop on the initial location l_0. The CUB-PTA conditions enforce a constraint $p_1 \leq p_2$. Therefore, the model is CUB-PTA for all valuations of p_1 and p_2 such that $p_1 \leq p_2$.

Contrarily, in Fig. 7.2d, the edge from the initial location l_0 to location l_1 (or location l_2) induces a constraint $\infty \leq p_2$ (respectively $\infty < p_1$, which is always false. Therefore, the model is not a CUB-PTA, whatever the valuation.

Finally, the example of Fig. 7.2b is CUB-PTA for all valuations of p.

Lemma 7.3.1. *Let $\mathcal{A}$ be a CUB-PTA. Let $v \models \mathcal{A}.K_0$ be a parameter valuation. Then $\mathcal{A}[v]$ is a CUB-TA.*

Proof. Let $v \models \mathcal{A}.K_0$. Let $e = (l, g, a, R, l')$ be an edge. Given a clock $x \in X$, from Definition 7.3.2, we have that $v \models \big(pub(I(l), x) \leq pub(g, x)\big)$, and therefore $pub(I(l), x)[v] \leq pub(g, x)[v]$. This matches the first case of Definition 7.3.1. The second case ($x \notin R$) is similar. ☐

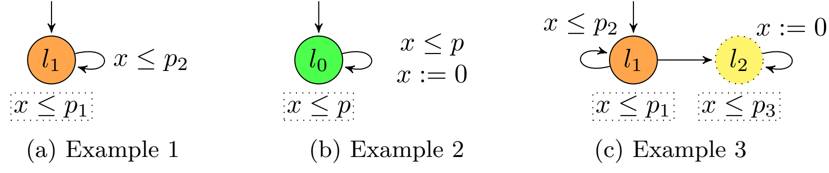
7.3.4 CUB PTA detection

Given an arbitrary PTA, our approach works as follows. Firstly, we check whether it is a CUB-PTA for some valuations. If it is, we proceed to the synthesis problem, using the cycle detection synthesis algorithm (Section 7.4); however, the result may be partial, as it will only be valid for the valuations for which the PTA is CUB. This incompleteness may come at the benefit of a more efficient synthesis. If it is CUB for no valuation, it has to be transformed into an equivalent CUB-PTA (which will be considered in Section 7.3.5).

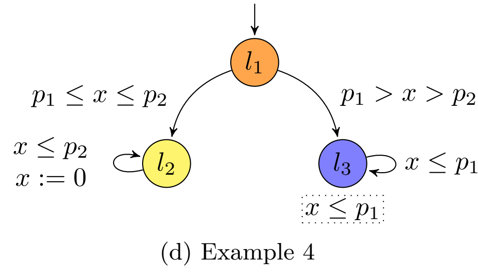
Algorithm 18: CUBdetect($\mathcal{A}$)

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, F, X, P, K_0, I, E)$
Output: A constraint K ensuring the PTA is a CUB-PTA

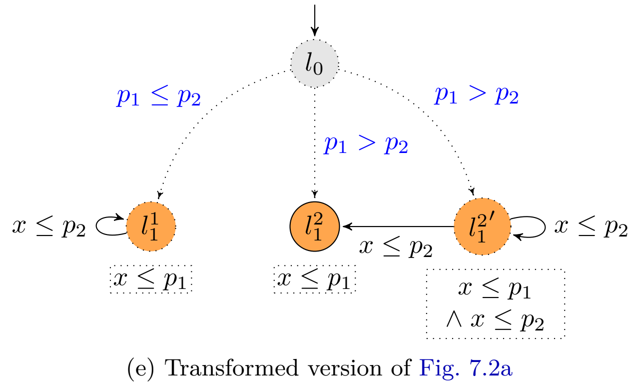
1 $K \leftarrow K_0$
2 **foreach** *edge* (l, g, a, R, l') **do**
3 **foreach** *clock* $x \in X$ **do**
4   $K \leftarrow K \wedge \big(pub(I(l), x) \leq pub(g, x)\big)$
5   **if** $x \notin R$ **then** $K \leftarrow K \wedge \big(pub(I(l), x) \leq pub(I(l'), x)\big)$
6 **return** K



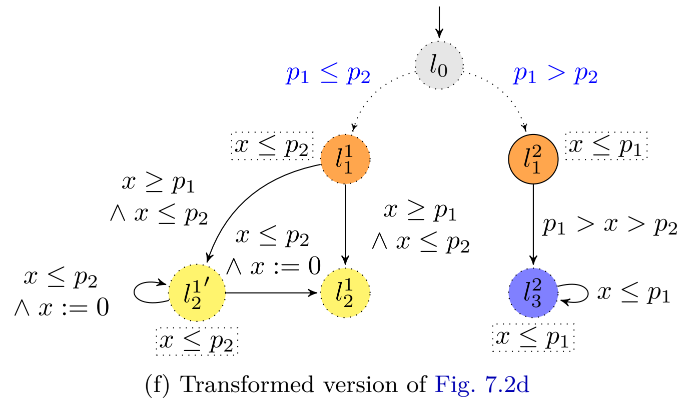
(a) Example 1 (b) Example 2 (c) Example 3



(d) Example 4



(e) Transformed version of Fig. 7.2a



(f) Transformed version of Fig. 7.2d

Figure 7.2 – Examples: detection of and transformation into CUB-PTAs

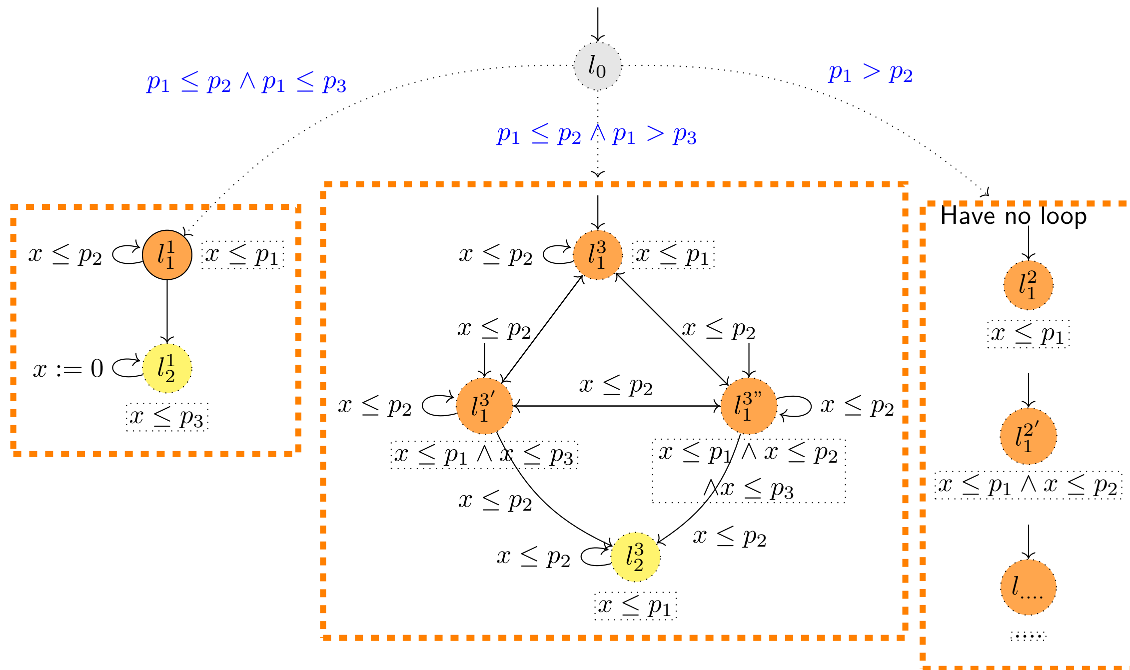
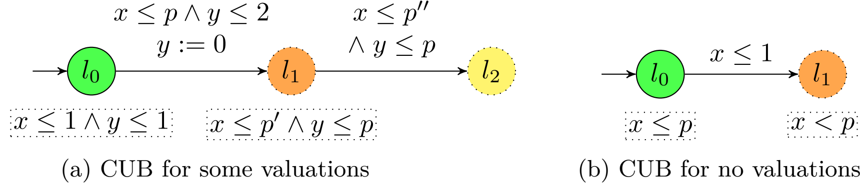


Figure 7.3 – Transformed version of Fig. 7.2c



(a) CUB for some valuations (b) CUB for no valuations

Figure 7.4 – Examples of PTAs to illustrate the CUB concept

Our procedure to detect whether a PTA is CUB for some valuations is given in Algorithm 18. For each edge in the PTA, we enforce the CUB condition on each clock by constraining the upper bound in the invariant of the source location to be smaller than or equal to the upper bound of the edge guard (line 4). Additionally, if the clock is not reset along this edge, then the upper bound of the source location invariant should be smaller than or equal to that of the target location (line 5). If the resulting set of constraints accepts parameter valuations (i. e. is not empty), then the PTA is a CUB-PTA for these valuations.

Example 26. Consider again the PTA $\mathcal{A}$ in Fig. 7.4a, assuming that $\mathcal{A}.K_0 = \top$. This PTA is CUB for $1 \leq p \wedge 1 \leq p' \wedge p' \leq p''$.

Example 27. Consider the PTA $\mathcal{A}$ in Fig. 7.4b, with $\mathcal{A}.K_0 = \top$. When handling location l_0 and clock x, line 4 yields $\mathcal{A}.K = \top \wedge [(p, \leq) \leq (1, \leq)] = p \leq 1$ and then, from line 5, $\mathcal{A}.K = p \leq 1 \wedge [(p, \leq) \leq (p, <)] = p \leq 1 \wedge p < p = \perp$. Hence, there is no valuation for which this PTA is CUB.

Proposition 7.3.2. *Let $\mathcal{A}$ be a PTA. Let $K = \mathsf{CUBdetect}(\mathcal{A})$. Then $\mathcal{A}(K)$ is a CUB-PTA.*

Proof. From the fact that Algorithm 18 gathers constraints to match Definition 7.3.2. □

7.3.5 Transforming a PTA into a disjunctive CUB-PTA

In this section, we show that an arbitrary PTA can be transformed into an extension of CUB-PTAs (namely *disjunctive CUB-PTA*), while preserving the symbolic runs.

For non-parametric TAs, it is shown in [WSW+15] that any TA can be transformed into an equivalent CUB-TA. This does not lift to CUB-PTAs.

Example 28. No equivalent CUB-PTA exists for the PTA in Fig. 7.2d where $K_0 = \top$. Indeed, the edge from l_1 to l_2 (resp. l_3) requires $p_1 \leq p_2$ (resp. $p_1 > p_2$). It is impossible to transform this PTA into a PTA where K_0 (which is $\top$) is included in both $p_1 \leq p_2$ and $p_1 > p_2$.

Therefore, in order to overcome this limitation, we propose an alternative definition of *disjunctive CUB-PTAs.* They can be seen as a union (as defined in the timed automata patterns of, e. g. [DHQ+08]) of CUB-PTAs.

Definition 7.3.3. A *disjunctive CUB-PTA* is a list of CUB-PTAs. Given a disjunctive CUB-PTA $\mathcal{A}_1, \dots, \mathcal{A}_n$, with $\mathcal{A}_i = (\Sigma_i, L_i, l_0^i, F_i X_i, P_i, K_0^i, I_i, E_i)$, the PTA *associated with* this disjunctive PTA is
$\mathcal{A} = (\bigcup_i \Sigma_i, \bigcup_i L_i \cup \{l_0\}, l_0, \bigcup_i F_i, \bigcup_i X_i, \bigcup_i P_i, \bigcup_i K_0^i, \bigcup_i I_i, E)$, where $E = \bigcup_i E_i \cup E'$ with $E' = \bigcup_i (l_0, K_0^i, \epsilon, X, l_0^i)$.

Basically, the PTA associated with a disjunctive CUB-PTA is just an additional initial location that connects to each of the CUB-PTAs initial locations, with its initial constraint on the guard.[2]

Example 29. In Fig. 7.2e (without the dotted, blue elements), two CUB-PTAs are depicted, one (say $\mathcal{A}_1$) on the left with locations superscripted by 1, and one (say $\mathcal{A}_2$) on the right superscripted with 2. Assume $\mathcal{A}_1.K_0$ is $p_1 \leq p_2$ and $\mathcal{A}_2.K_0$ is $p_1 > p_2$. Then the full Fig. 7.2e (including dotted elements) is the PTA associated with the disjunctive CUB-PTA made of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Technically, the key idea behind the transformation from a TA into a CUB-TA in [WSW+15] is as follows: whenever a location l is followed by an edge e and a location l' for which $ub(g, x) < ub(l, x)$ or $ub(l', x) < ub(l, x)$ for some x if $x \notin R$, otherwise $ub(g, x) < ub(l, x)$, location l is split into two locations: one (say l_1) with a "decreased upper bound", i. e. $x \lhd ub(l', x)$, that is then connected to l'; and one (say l_2) with the same invariant as in l, and with no transition to l'. Therefore, the original behavior is maintained. Note that, this transformation induces some non-determinism (one must non-deterministically choose whether one enters l_1 or l_2, which will impact the future ability to enter l') but this has no impact on the existence of a non-Zeno cycle.

Here, we extend this principle to CUB-PTAs. A major difference is that, in the parametric setting, comparing two clock upper bounds does not give a Boolean answer but a parametric answer. For example, in a TA, $(2, \leq) \leq (3, <)$ holds (this is true), whereas in a PTA $(p_1, \leq) \leq (p_2, <)$ denotes the *constraint* $p_1 < p_2$. Therefore, the principle of our transformation is that, whenever we have to compare two parametric clock upper bounds, we consider both cases: here either $p_1 < p_2$ (in which case the first location does not need to be split) or $p_1 \geq p_2$ (in which case the first location shall be split). This yields a finite list of CUB-PTAs: each of these CUB-PTAs consists in one particular ordering of all parametric linear terms used as upper bounds in guards and invariants. (In practice, in order to reduce the complexity, we only define an order on the parametric linear terms the comparison of which is needed during the transformation process.)

Example 30. Let us transform the PTA in Fig. 7.2a: if $p_1 \leq p_2$ then the PTA is already CUB, and l_1 does not need to be split. This yields a first CUB-PTA, depicted on the left-hand side of Fig. 7.2e. However, if $p_1 > p_2$, then l_1 needs to be split into $l_1^{2'}$ (where time cannot go beyond p_2) and into l_1^2 (where time can go beyond p_2, until p_1), but the self-loop cannot be taken anymore (otherwise

[2] A purely parametric constraint (e. g. $p_1 > p_2 \land p_3 = 3$) is generally not allowed by the PTA syntax, but can be simulated using appropriate clocks (e. g. $p_1 > x > p_2 \land p_3 = x' = 3$). Such parametric constraints are allowed in the input syntax of IMITATOR.

the associated guard makes the PTA not CUB). This yields a second CUB-PTA, depicted on the right-hand side of Fig. 7.2e. Both make a disjunctive CUB-PTA equivalent to Fig. 7.2a.

Similarly, we give the transformation of Fig. 7.2d in Fig. 7.2f, and Fig. 7.2c in Fig. 7.3.

Let us now show that any PTA can be transformed into a disjunctive CUB-PTA. A PTA is not a CUB-PTA if there exists an edge entailing a decreasing upper bound for some clock: this is a “problematic” edge. A problematic edge $e = (l, g, a, R, l')$ is detected by comparing the upper bounds of location l, guard g and location l'. Unfortunately, when these upper bounds contain parameters, their comparison might be impossible, so all cases have to be considered. Since all problematic edges can be detected, any PTA can easily be transformed into a disjunctive CUB-PTA by splitting locations and enforcing their invariants to cater for all possible cases. A new initial location is created, that is connected to these copies. Thus, all problematic edges are removed, and the transformed model is a disjunctive CUB-PTA.

Algorithm 19 presents the transformation of a PTA into a disjunctive CUB-PTA. It comprises five parts preceded by an initialization phase. Note that, in order to reduce the state space explosion and to improve efficiency, the algorithm generates only the necessary parameter relations and operates on-the-fly:

0. Initially, the automaton considered is the input one, and an empty set of constraints is associated with each location, that will be modified by the algorithm. The pair composed of automaton and constraints is added to a queue of elements to be handled until all of them have been considered;
1. The first part (lines 4–33) splits the different constraints cases for the problematic edges. For each of these, K (resp. θ) gathers the parametric (resp. timed) constraint w.r.t. the edge.
2. In lines 35–38, a location is created for each constraint in $constraints_q(l)$. Its associated invariant is a conjunction of K and $I(l)$ in L' and $cub_ls(l)$.
3. Then, new edges are created to connect these new locations (lines 39–42) ;
4. Finally, in lines 43–47, all problematic edges are deleted.
5. At the end of the loop (line 48), $\mathcal{A}_q$ is a CUB-PTA that is added to the disjunctive CUB-PTA $\mathcal{D}$. When exiting the loop, the initial state l_0 is linked to all initial states of CUB-PTAs before $\mathcal{D}$ is returned.

7.4 Zeno-free cycle synthesis in CUB-PTAs

Taking a disjunctive CUB-PTA as input, we show in this section that synthesizing the parameter valuations for which there exists at least one non-Zeno cycle (and therefore an infinite non-Zeno run) reduces to a SCC (strongly connected component) synthesis problem.

Algorithm 19: CUBtrans($\mathcal{A}$): Transformation into a CUB-PTA

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, K_0, I, E)$
Output: PTA $\mathcal{D}$ associated with a disjunctive CUB-PTA

1 $\mathcal{A}_0 \leftarrow \mathcal{A}$; $\forall l \in L, constraints_0(l) \leftarrow \emptyset$;
2 *queue* $\mathcal{Q} \leftarrow \langle \mathcal{A}_0, constraints_0 \rangle$;
3 **while** $\mathcal{Q} \neq \emptyset$ **do**
4 $\quad \langle \mathcal{A}_q(\Sigma, L', l'_0, X, P, K'_0, I, E'), constraints_q \rangle \leftarrow dequeue(\mathcal{Q})$;
5 $\quad Kadding \leftarrow \top$;
6 $\quad$ **while** $Kadding = \top$ **do**
7 $\quad\quad Kadding \leftarrow \bot$;
8 $\quad\quad$ **foreach** *edge* $(l, g, a, R, l') \in E'$ **do**
9 $\quad\quad\quad K \leftarrow \bigwedge_{x \in X} \Big[pub(I(l), x) \leq pub(g, x) \wedge$
10 $\quad\quad\quad\quad$ **if** $x \notin R$ **then** $pub(I(l), x) \leq pub(I(l'), x)$ **else** $\top \Big]$;
11 $\quad\quad\quad \theta \leftarrow \bigwedge_{x \in X} \Big[(I(l), x) \wedge (g, x) \wedge$ **if** $x \notin R$ **then** $(I(l'), x)$ **else** $\top \Big]$;
12 $\quad\quad\quad$ **if** $\theta \notin constraints_q(l)$ **then**
13 $\quad\quad\quad\quad$ **if** K *is* $\bot$ **then**
14 $\quad\quad\quad\quad\quad constraints_q(l) \leftarrow constraints_q(l) \cup \{\theta\}$;
15 $\quad\quad\quad\quad$ **if** K *is a parameter constraint and* $K \wedge K'_0 \neq \emptyset$ **then**
16 $\quad\quad\quad\quad\quad$ **if** $(\neg K \wedge K'_0) \neq \emptyset$ **then**
17 $\quad\quad\quad\quad\quad\quad \mathcal{A}' \leftarrow (\Sigma, L', l'_0, X, P, (\neg K \wedge K'_0), I, E')$;
18 $\quad\quad\quad\quad\quad\quad \mathcal{Q} \leftarrow \langle \mathcal{A}', constraints_q(l) \cup \{\theta\} \rangle$;
19 $\quad\quad\quad\quad\quad K'_0 \leftarrow K'_0 \wedge K$;
20 $\quad\quad\quad$ **foreach** *constraint* K *in* $constraints_q(l)$ **do**
21 $\quad\quad\quad\quad K' \leftarrow \bigwedge_{x \in X} \Big[pubI(l), x) \leq pub(g, x) \wedge$
22 $\quad\quad\quad\quad\quad$ **if** $x \notin R$ **then** $pub(I(l), x) \leq pub(k, x)$ **else** $\top \Big]$;
23 $\quad\quad\quad\quad \theta' \leftarrow \bigwedge_{x \in X} \Big[(I(l), x) \wedge (g, x) \wedge$ **if** $x \notin R$ **then** (K, x) **else** $\top \Big]$;
24 $\quad\quad\quad\quad$ **if** $\theta' \notin constraints_q(l)$ **then**
25 $\quad\quad\quad\quad\quad$ **if** K' *is* $\bot$ **then**
26 $\quad\quad\quad\quad\quad\quad constraints_q(l) \leftarrow constraints_q(l) \cup \{\theta'\}$;
27 $\quad\quad\quad\quad\quad$ **if** K' *is a parameter constraint and* $K' \wedge K'_0 \neq \emptyset$ **then**
28 $\quad\quad\quad\quad\quad\quad$ **if** $(\neg K' \wedge K'_0) \neq \emptyset$ **then**
29 $\quad\quad\quad\quad\quad\quad\quad \mathcal{A}' \leftarrow (\Sigma, L', l'_0, X, P, (\neg K' \wedge K'_0), I, E')$;
30 $\quad\quad\quad\quad\quad\quad\quad \mathcal{Q} \leftarrow \langle \mathcal{A}', constraints_q(l) \cup \{\theta'\} \rangle$;
31 $\quad\quad\quad\quad\quad\quad K'_0 \leftarrow K'_0 \wedge K'$;
32 $\quad\quad\quad$ **if** $constraints_q(l)$ *is increased* **then**
33 $\quad\quad\quad\quad Kadding \leftarrow \top$;
34 $\quad cub_ls \leftarrow \emptyset$;
35 $\quad$ **foreach** *constraint* K *in* $constraints_q(l)$ **do**
36 $\quad\quad$ create a new location cub_l;
37 $\quad\quad I(cub_l) \leftarrow K \wedge I(l)$; $L' \leftarrow L' \cup \{cub_l\}$;
38 $\quad\quad cub_ls \leftarrow cub_ls \cup \{cub_l\}$;
39 $\quad$ **foreach** *location* cub_l *in* cub_ls **do**
40 $\quad\quad$ **foreach** *edge* (l', g, a, R, l) *or* (l, g, a, R, l') *respectively with* $l' \in L'$ **do**
41 $\quad\quad\quad E' \leftarrow (l', g, a, R, cub_l)$ or (cub_l, g, a, R, l');
42 $\quad\quad\quad E' \leftarrow (cub_l', g, a, R, cub_l)$ or (cub_l, g, a, R, cub_l') with $cub_l' \in cub_ls$;
43 $\quad$ **foreach** *edge* $e(l, g, a, R, l') \in E'$ **do**
44 $\quad\quad K \leftarrow \bigwedge_{x \in X} \Big[pub(I(l), x) \leq pub(g, x) \wedge$
45 $\quad\quad\quad$ **if** $x \notin R$ **then** $pub(I(l), x) \leq pub(I(l'), x)$ **else** $\top \Big]$;
46 $\quad\quad$ **if** $K = \bot$ *or* (K *is a parameter constraint and* $K \wedge K'_0 = \emptyset$) **then**
47 $\quad\quad\quad E' \leftarrow E' \setminus \{e\}$
48 $\quad \mathcal{D} \cup \mathcal{A}_q$;
49 **foreach** $\mathcal{A}_q(\Sigma, L', l'_0, X, P, K'_0, I, E') \in \mathcal{D}$ **do**
50 $\quad$ add edge from l_0 of $\mathcal{D}$ to set of $cub_ls \in l'_0 \cup l'_0$ with K'_0 as guard;
51 **return** $\mathcal{D}$;

First, we define a light extension of the parametric zone graph as follows. The *extended parametric zone graph* of a PTA $\mathcal{A}$ is identical to its parametric zone graph, except that any transition $(\mathbf{s}, e, \mathbf{s}')$ is replaced with $(\mathbf{s}, (e, b), \mathbf{s}')$, where b is a Boolean flag which is true if time can *potentially* elapse between $\mathbf{s}$ and $\mathbf{s}'$. In practice, b can be computed as follows, given $\mathbf{s} = (l, C)$ and edge e:

1. add a fresh extra clock x_0 to the constraint C, i. e. compute $C \wedge x_0 = 0$
2. compute the successor $\mathbf{s}' = (l', C')$ of $(l, C \wedge x_0 = 0)$ via edge e
3. check whether $C' \Rightarrow x_0 = 0$: if so, then $b = \texttt{false}$; otherwise $b = \texttt{true}$.

Introducing such a clock is cheap: the check is not expensive, and the extra clock does not impact the size of the parametric zone graph:

As mentioned in [WSW+15], introducing the clock x_0 here is different from the approach of introducing an extra clock for non-Zenoness detection [Tri99] as x_0 is 0 in all nodes of the zone graph and can be eliminated from the memory, therefore not requiring more space nor extra states.

In contrast to non-parametric TAs, the flag b does not necessarily mean that time can necessarily elapse for *all* parameter valuations. Consider the example in Fig. 7.2b. After taking one loop, we have that $x_0 \leq p$: therefore, x_0 is not necessarily 0, and b is $\texttt{true}$. But consider v such that $v(p) = 0$: then in l_1 time can never elapse.

However, we show in the following lemma that the flag b *does* denote time elapsing for *strictly positive* parameters. But let us first recall a lemma used in it, relating concrete and symbolic runs.

Lemma 7.4.1. *Let $\mathcal{A}$ be a PTA, and let $\mathbf{r}$ be a symbolic run of $\mathcal{A}$ reaching (l, C). Let $v \models \mathcal{A}.K_0$ be a parameter valuation. There exists an equivalent concrete run in $\mathcal{A}[v]$ iff $v \models C{\downarrow_P}$.*

'

Proof. From [HRSV02, Propositions 3.17 and 3.18]. ☐

Given a symbolic run $\mathbf{r}$ reaching (l, C), we call the *concrete runs associated with* $\mathbf{r}$ the concrete runs equivalent to $\mathbf{r}$ in $\mathcal{A}[v]$, for all $v \models C{\downarrow_P}$.

Lemma 7.4.2. *Let $(l, C) \overset{e,b}{\Rightarrow} (l', C')$ be a transition of the extended parametric zone graph of a PTA $\mathcal{A}$. Then, for any strictly positive parameter valuation in $C'{\downarrow_P}$, there exists an equivalent transition in $\mathcal{A}[v]$ in which time can elapse.*

Proof. First note that, for any $v \models C'{\downarrow_P}$, an equivalent concrete transition exists in $\mathcal{A}[v]$, from Lemma 7.4.1. Now, since b is true, the extra clock x_0 in the state of the extended parametric zone graph corresponding to (l, C') is either unbounded, or bounded by some parametric linear term *plt*. If it is unbounded, then time can elapse for any valuation, and the lemma holds trivially. Assume $x_0 \leq \mathit{plt}$ for some *plt*. As our parameters are strictly positive, then for any valuation v, $\mathit{plt}[v]$

evaluates to a strictly positive rational, and therefore time can elapse along this transition in $\mathcal{A}[v]$. □

Definition 7.4.1. An infinite symbolic run $\mathbf{r}$ is non-Zeno if all its associated concrete runs are non-Zeno.

In the remainder of this section, given an edge $e = (l, g, a, R, l')$, $e.R$ denotes that the clocks in R reset along e.

The following theorem states that an infinite symbolic run is non-Zeno iff the time can (potentially) elapse along infinitely many edges and, whenever a clock is bounded from above, then eventually either this clock is reset or it becomes unbounded.

Theorem 7.4.3. *Let* $\mathbf{r} = \mathbf{s}_0 \overset{(e_0,b_0)}{\Rightarrow} \mathbf{s}_1 \overset{(e_1,b_1)}{\Rightarrow} \cdots$ *be an infinite symbolic run of the extended parametric zone graph of a CUB-PTA* $\mathcal{A}$. $\mathbf{r}$ *is non-Zeno if and only if*

- $*$ *there exist infinitely many* k *such that* $b_k = \mathtt{true}$*; and*
- $\star$ *for all* $x \in X$*, for all* $i \geq 0$*, given* $\mathbf{s}_i = (l_i, C_i)$*, if* $pub(l_i, x) \neq \infty$*, there exists* j *such that* $j \geq i$ *and* $x \in e_j.R$ *or* $pub(l_j, x) = \infty$*.*

Proof. $\Rightarrow$ If $\mathbf{r}$ is non-Zeno, $*$ is trivially true. Consider the case when a clock x is bounded from above (i. e. with a parametric upper bound other than ∞): for any parameter valuation, x is bounded from above (even for arbitrarily large valuations). Therefore the clock x must be reset later, or the upper bound becomes infinity since by definition its value goes unbounded along the run; otherwise, we have an empty symbolic state and thus an infeasible run. Hence, $\star$ holds.

$\Leftarrow$ In the following, we show that if $*$ and $\star$ are true, then $\mathbf{r}$ is non-Zeno. Let the following be a segment of $\mathbf{r}$ according to $*$ and $\star$:

$$(l_i, C_i) \overset{(e_i,b_i)}{\Rightarrow} (l_{i+1}, C_{i+1}) \overset{(e_{i+1},b_{i+1})}{\Rightarrow} \cdots (l_j, C_j) \overset{(e_j,b_j)}{\Rightarrow} \cdots (l_k, C_k) \overset{(e_k,b_k)}{\Rightarrow} \cdots$$

where $i \leq j \leq k$ and $b_j = \mathtt{true}$. Furthermore, for all $x \in X$, if $pub(l_j, x) \neq \infty$, there exists m, n such that $i - 1 \leq m < j \leq n \leq k - 1$ such that $x \in e_m.R$ (or $m = -1$, i. e. x is "reset" before the initial state as all clocks are initially 0) and, $x \in e_n.R$ or $pub(l_n, x) = \infty$ (from condition $\star$). That is, the segment contains a transition which can be delayed locally. Furthermore, the segment covers the "life-span" (between two resets) of all clocks in l_j which have an upper bound other than ∞.

The infinite symbolic run $\mathbf{r}$ is progressive if and only if it takes an unbounded amount of time for any parameter valuation. Since there are infinitely many segments as above in $\mathbf{r}$, if any such segment can take a positive amount of time for all its valuations, then the run $\mathbf{r}$ is progressive and thus non-Zeno, and then non-Zeno for any valuation (from Definition 7.4.1). Next, we show that the segment can take a positive amount of time.

Note that, because b_j is true, then from Lemma 7.4.2, for any strictly positive valuation in $C_n{\downarrow_P}$, the time than *can* elapse from l_j to l_{j+1} is strictly positive. Furthermore, since $\mathcal{A}$ is a CUB-PTA, therefore for any strictly positive valuation $v \models C_n{\downarrow_P}$, from Lemma 7.3.1 $\mathcal{A}[v]$ is a CUB-TA. From the syntax of CUB-TAs, a run cannot be blocked in the future due to a too small clock upper bound, as clock upper bounds are always non-decreasing. Therefore, for any strictly positive valuation in $C_n{\downarrow_P}$, the parametric time than can elapse from l_m to l_n is strictly positive. Therefore, for any strictly positive valuation $v \models C_n{\downarrow_P}$, there exists an equivalent concrete run in $\mathcal{A}[v]$ that is progressive. Hence $\mathbf{r}$ is progressive. With the arguments above, we conclude that the theorem holds.

□

We now show in the following that synthesizing parameter valuations for which there exists a non-Zeno infinite run reduces to a SCC search problem.

First, given a SCC *scc*, we denote by *scc.K* the parameter constraint associated with *scc*, i. e. $C{\downarrow_P}$, where (l, C) is any state of the SCC.[3]

Theorem 7.4.4. *Let $\mathcal{A}$ be a CUB-PTA of finite extended parametric zone graph $\mathcal{G}$. Let v be a strictly positive parameter valuation. $\mathcal{A}[v]$ contains a non-Zeno infinite run if and only if $\mathcal{G}$ contains a reachable SCC scc such that $v \models scc.K$ and*

- † *scc contains a transition $\mathbf{s} \overset{(e,b)}{\Rightarrow} \mathbf{s}'$ such that $b = \texttt{true}$; and*
- ‡ *for every clock x in X, given $\mathbf{s} = (l, C)$, if $pub(l, x) \neq \infty$ for some state $\mathbf{s}$ in scc, there exists a transition in scc with label (e, b) such that $x \in e.R$.*

Proof. ⇒ Assume $\mathcal{A}[v]$ contains a non-Zeno infinite run. From Lemma 7.4.1, there exists an equivalent symbolic run $\mathbf{r}$ in $\mathcal{G}$. Since $\mathcal{G}$ is finite-state, $\mathbf{r}$ must visit a set of states and transitions, denoted as *Inf*, infinitely often. There must be a SCC, say *scc*, which contains *Inf*. *Inf* must contain a transition with a label b being true (by contradiction) and therefore † is trivially true.

Next, we prove ‡ by contradiction. Assume there is a state $\mathbf{s}$ in *scc* where a clock x has an upper bound *plt* which is not ∞ and there is no transition in *scc* which resets x. Because the upper bound of x never decreases (by definition of CUB-PTAs), the upper bound of x at every state in *scc* must be equal to *plt*. Since *scc* contains *Inf*, this implies that $\mathbf{r}$ is Zeno as x is always bounded from above and never reset, which contradicts our assumption that $\mathbf{r}$ is non-Zeno. Thus, *scc* must satisfy ‡.

⇐ Assume there is a SCC in $\mathcal{G}$ satisfying † and ‡. Let $\mathbf{r}$ be a symbolic run which visits every state/transition in the SCC infinitely often. It is easy to

[3]Following a well-known result for PTAs, all symbolic states belonging to a same cycle in a parametric zone graph have the same parameter constraint.

see that $\mathbf{r}$ satisfies $*$ of Theorem 7.4.3 because of †. By ‡, we conclude that every clock which has an upper bound other than ∞ at a state is reset later. Therefore, $\mathbf{r}$ is non-Zeno by Theorem 7.4.3. From Definition 7.4.1, any concrete run associated with $\mathbf{r}$ is non-Zeno; from Lemma 7.4.1, the parameter valuations having such an equivalent run are exactly $scc.K$. Therefore, we conclude that the theorem holds.

□

Therefore, from Theorem 7.4.4, synthesizing valuations yielding an infinite symbolic run reduces to a SCC search problem in the extended parametric zone graph. Then, we need to test each SCC against two conditions: whether it contains a transition which can be locally delayed (i. e. whether it contains a transition where $b = \texttt{true}$); and whether every clock having an upper bound other than ∞ at some state is reset along some transition in the SCC. Then, for all SCCs matching these two conditions, we return the associated parameter constraint.

Algorithm 20, synthNZ, solves the non-Zeno synthesis problem for CUB-PTAs. synthNZ simply iterates on the SCCs, and gathers their associated parameter constraints whenever they satisfy the conditions in Theorem 7.4.4.

Algorithm 20: CUB-PTA non-Zeno synthesis algorithm synthNZ($\mathcal{A}$)

Input: CUB-PTA $\mathcal{A}$ and its extended parametric zone graph $\mathcal{G}$
Output: constraint K_{NZ} gathering valuations for which there is a non-Zeno infinite run

```
1 K_NZ ← ⊥ while there are unvisited states in G do
2     find a new SCC scc
3     mark all states in scc as visited
4     if scc satisfies † and ‡ then
5         K_NZ ← K_NZ ∨ scc.K
6 return K_NZ
```

If the extended parametric zone graph $\mathcal{G}$ is finite, then the correctness and completeness of synthNZ immediately follow from Theorem 7.4.4. If only an incomplete part of $\mathcal{G}$ is computed (e. g. by bounding the exploration depth, or the number of explored states, or the execution time) then only the $\Leftarrow$ direction of Theorem 7.4.4 holds: in that case, the result of synthNZ is correct but non-complete, i. e. it is a valid under-approximation. In the context of parametric model checking, knowing which parameter valuations violate the property is already very helpful to the designer, as it helps to discard unsafe valuations, and to refine the model.

Example 31. The parametric zone graph example in Figure 7.6a is generated after detecting Figure 7.2a as a partial CUB-PTA, which is CUB-PTA for only parameter valuations such that $p_1 \leq p_2$ (missing parameter valuations $p_1 > p_2$ will be checked with the disjunctive CUB-PTA). After taking one loop, we have

$x \leq p_1$: therefore, x is bounded and b is `false`. Consequently, the result returned is false which means that this loop contains a Zeno run for all valuations such that $p_1 \leq p_2$. Otherwise, in Figure 7.2b (already CUB-PTA for all parameter valuations), x is reset on the loop and then b is `true`. Note that x is not necessarily 0 but consider v such that $v(p) = 0$: then in l_1 time can never elapse.

Similarly, we give the parametric zone graph of disjunctive CUB-PTA of Fig. 7.3 in Fig. 7.6b, where non-Zeno runs locate in constraints $p_1 \leq p_2 \wedge p_1 \leq p_3$ and $p_1 \leq p_2 \wedge p_1 > p_3$ (or $p_1 \leq p_2$).

Finally, the flowchart in Fig. 7.5 gives an overview how the non-Zeno synthesis process works.

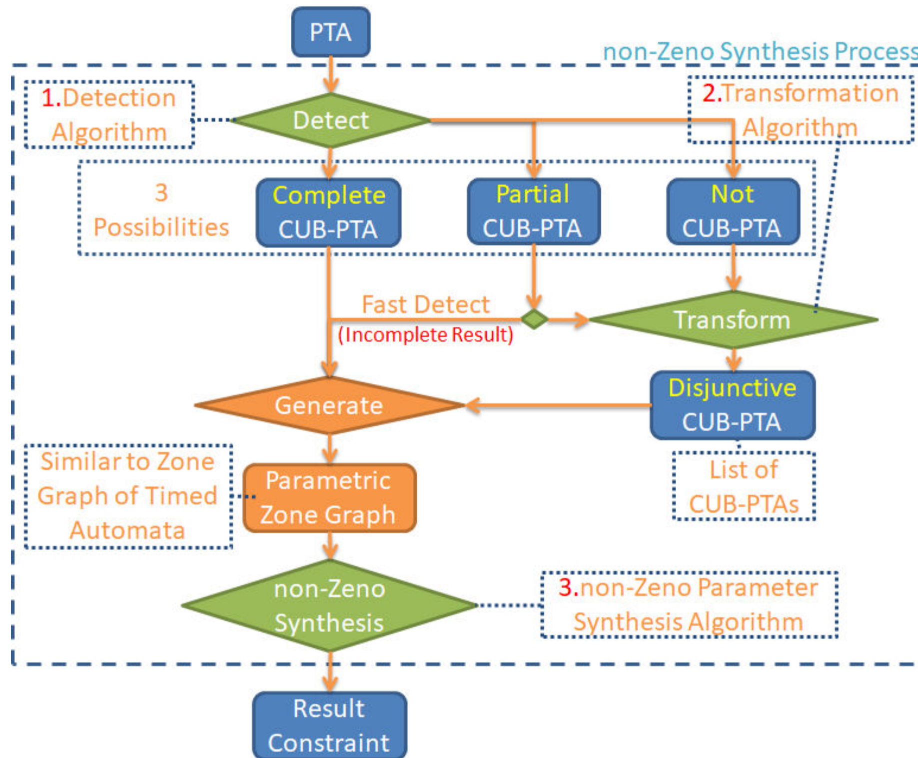
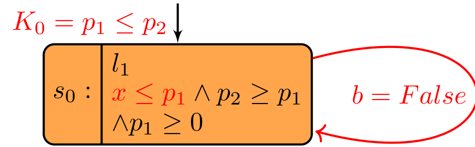
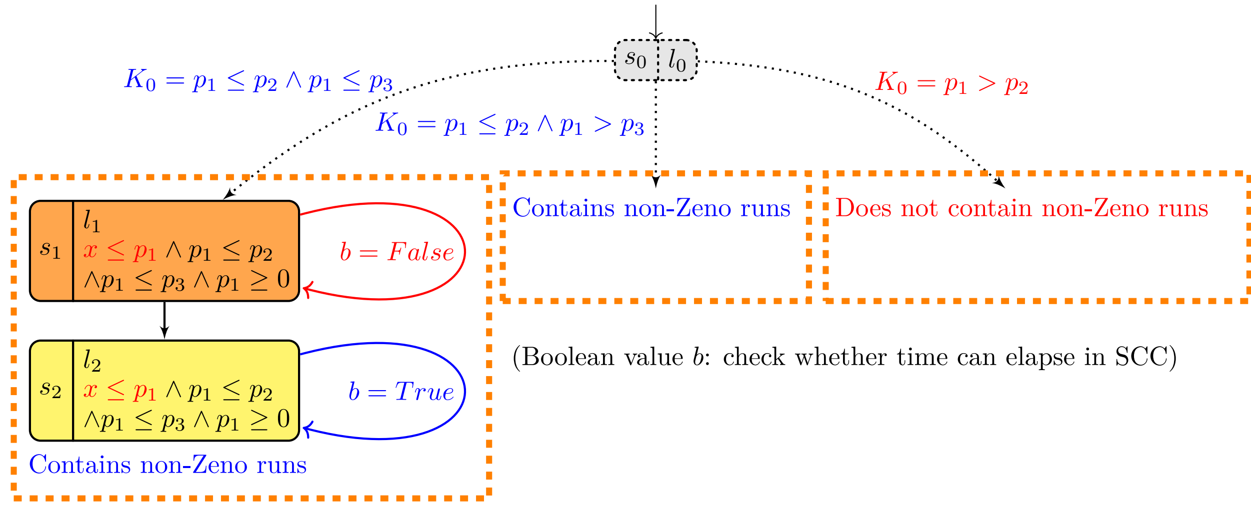


Figure 7.5 – Non-Zeno synthesis process flowchart



Emptiness non-Zeno check: False

(a) Parametric zone graph of the partial CUB-PTA of Figure 7.2a



Emptiness non-Zeno check: True for $p_1 \leq p_2$

(b) Parametric zone graph of the disjunctive CUB-PTA of Figure 7.3

Figure 7.6 – Parametric zone graph examples of the disjunctive CUB-PTAs

7.5 Distributing non-Zeno parametric model checking

A problem of the CUB-PTA approach is the number of CUB-PTAs in a disjunctive CUB-PTA might increase when the number of constants and parameters of PTA increases. Although the transforming of disjunctive CUB-PTA is quite fast with such PTAs, the size of the transformed disjunctive CUB-PTA can be costly for our non-Zeno parameter synthesis algorithm. In order to verify more complex and larger case studies, especially for PTAs containing many constants and parameters, we will therefore distribute our non-Zeno parameter synthesis algorithm on clusters. By splitting a disjunctive CUB-PTA into a set of CUB-PTAs, each node can call the non-Zeno parameter synthesis algorithm in Algorithm 20 on each CUB-PTA independently, thanks to the splittable structure of disjunctive CUB-PTA. After calling the algorithm, the node will return a constraint as a result. In the end, to have a final resulting constraint, a node will have the responsibility to gather and conjoin all constraints received from all nodes to where the CUB-PTAs are distributed.

In this work, we reuse the master-worker scheme from Section 3.3: the workers ask the master for CUB-PTAs on which they call non-Zeno parameter synthesis algorithm in Algorithm 20. Finally, the workers send the corresponding result parameter constraints back to the master. Moreover, the master does not synthesize by itself, but only distributes the work.

In the following sections, we formalize the abstract algorithms for the master and workers. The experimental validation for this distributed algorithm will be reported later in Section 7.6.2.

7.5.1 Master algorithms

The master algorithm is given in Algorithm 21. Three variables are maintained throughout the master's algorithm: the set $\mathcal{L}$ of CUB-PTAs at line 1, the final constraint K containing non-Zeno runs at line 5 and the *counter* variable at line 6 used for terminating the master node.

At line 1, the input disjunctive CUB-PTA $\mathcal{A}$ is split into a set of CUB-PTAs $\mathcal{A}$s, which are stored in set $\mathcal{L}$. Depending on the number of CUB-PTAs (N_{PTA}) in $\mathcal{L}$, the master can determine maximum required nodes N_{proc} at line 3, which is equal to $N_{PTA} + 1$ (the number of workers plus the master node). After that, at line 4, the master sends to each worker n a CUB-PTA in $\mathcal{L}$, thanks to the sub-function $sendPTA(n)$ from line 16. In the sub-function, the master simply picks a CUB-PTA from the set $\mathcal{L}$, and then sends it to worker n using the tagged message PTA(a) with a is a CUB-PTA in $\mathcal{L}$. Note that, the number of nodes N_{proc} can be smaller than the number of PTAs N_{PTA} so that the master cannot send all PTAs in $\mathcal{L}$ to all workers at once. Therefore, the master will wait for some workers finishing synthesizing on their CUB-PTAs, in order to continue sending the remaining CUB-PTAs in $\mathcal{L}$. At lines 5 and 6, the final constraint K is initialized to false constraint denoted $\perp$, and the variable *counter* is initialized

to the number of workers which equals $N_{proc} - 1$. After sending a CUB-PTA to workers at line 16, in the while loop at line 7, the blocking function *receive*() is used to receive back a constraint K_{NZ} from the worker n. The constraint K_{NZ} received by the master, will then be conjoined with the current constraint K to have the new final constraint. At line 10, in case there still remains CUB-PTAs in $\mathcal{L}$, the master again sends a remaining CUB-PTA to the worker n until $\mathcal{L}$ becomes empty, thanks to the while loop at line 7. Otherwise from line 12, when $\mathcal{L}$ is empty, the master sends a STOP() tagged message to terminate the worker n along with counting down the *counter* variable by one unit. When the *counter* variable counts to zero which means all workers are terminated, the master will exit the loop at line 7, and the final constraint K containing non-Zeno runs will be returned.

For simplicity purposes, the set $\mathcal{L}$ is described as a set of CUB-PTAs and *sendPTA*(n) function allows sending a CUB-PTA. In our implementation, sending a whole CUB-PTA is costly. Thus $\mathcal{L}$ stores only hash keys of CUB-PTAs and *sendPTA*(n) function only sends an integer value.

7.5.2 Worker algorithm

The main responsibility of the workers is to call the non-Zeno synthesis algorithm in Algorithm 20. As a consequence, the worker cannot guarantee its termination due to the proof in Section 7.2. If one of the workers cannot terminate, it makes the master cannot terminate as well. This problem will be discussed later in this section but first, let us assume that all workers will terminate eventually.

The worker algorithm is given in Algorithm 22 which is made of a simple loop within the blocking function *receive*() inside, which is a conditional switch statement to determine what to do with each specific tagged message from the master. In the switch statement, there are two cases for two specific tagged messages. The first tag at line 3 asks the worker to terminate immediately. The second one at line 4, indicates the worker receives a tagged message PTA(a) containing a CUB-PTA a. After that, the worker will call Algorithm 20 on the CUB-PTA a, and then send back a parametric constraint K_{NZ} to the master m by using the RESULT(K_{NZ}) tagged message at line 6.

In case one of worker does not terminate, at line 5, the worker will be stuck in calling Algorithm 20 and then cannot terminate. To solve this problem, a depth limit of state exploration is set to guarantee the termination of the worker at a predefined depth. Note that, if the depth limit is reached, the return result of the worker will be an under-approximation. This interception technique will not be mentioned in our algorithms to avoid unnecessary complications.

Example 32. Consider again the disjunctive CUB-PTA example in Fig. 7.2e. Assume this example is distributed on 4 processes (nodes). In the beginning, the master splits the disjunctive CUB-PTA into 3 CUB-PTAs corresponding to 3 orange rectangles as depicted on the figure. The number of required processes for this example is 4 (one for master, and three others for workers to work on these CUB-PTAs). After that, the master sends these CUB-PTAs which

Algorithm 21: distSynthNZ($\mathcal{A}$): Master

```
    Input: Disjunctive CUB-PTA A, a number of available processes N
    Output: constraint K gathering valuations for which there is a
            non-Zeno infinite run
    // Initialization phase
 1  L ← split disjunctive CUB-PTA A into a set of CUB-PTA As
 2  N_PTA ← |L|
    // Master index is 1
 3  N_proc ← if N > N_PTA then N_PTA + 1 else N
 4  foreach process n ∈ {2, ..., N_proc} do sendPTA(n)
 5  K ← ⊥
    // counter is equal to the number of workers
 6  counter ← N_proc − 1
    // Main phase
 7  while counter > 0 do
 8  |   n, RESULT(K_NZ) ← receive
 9  |   K ← K ∪ {K_NZ}
10  |   if L ≠ ∅ then
11  |   |   sendPTA(n)
12  |   else
13  |   |   counter ← counter − 1
14  |   |   send(n, STOP())
    // Finalization phase
15  return K

16  function sendPTA(n) begin
17  |   pick a ∈ L
18  |   L ← L \ {a}
19  |   send(n, PTA(a))
```

have initial constraints $K_0 = p_1 \leq p_2 \wedge p_1 \leq p_3$, $K_0 = p_1 \leq p_2 \wedge p_1 \geq p_3$ and $K_0 = p_1 \geq p_2$ to worker 1, worker 2 and worker 3 respectively. In the CUB-PTA having the constraint $K_0 = p_1 \geq p_2$, which has not any infinite run. Then the worker 3 sends back a false constraint to the master. Otherwise, non-Zeno runs are detected in the other CUB-PTAs and the worker 1 and worker 2 send back their constraints $p_1 \leq p_2 \wedge p_1 \leq p_3$ and $p_1 \leq p_2 \wedge p_1 \geq p_3$ to the master respectively. At the master side, all constraints of workers are gathered and conjoined. After that, the master checks for a remaining CUB-PTA to send to the worker, but in this case, there is no remaining CUB-PTA left. Thus, the master sends stop tagged message to terminate each worker. Finally, the master returns the final constraint $p_1 \leq p_2$.

Algorithm 22: distSynthNZ($\mathcal{A}$): worker n

Input: Disjunctive CUB-PTA $\mathcal{A}$, a number of available processes N
Output: constraint K_{NZ} gathering valuations for which there is a non-Zeno infinite run

```
1 while true do
2     switch receive do
3         case m, STOP: do  return false
4         case m, PTA(a): do
5             K_NZ ← synthNZ(a) (Algorithm 20)
6             send(m, RESULT(K_NZ))
```

7.5.3 Handling the case of a network of PTAs

We briefly discuss in the following how we handle synchronization of PTAs in our distributed algorithms. Given a network of PTAs,

1. Each PTA will first be transformed into a disjunctive CUB-PTA. Then a network of PTAs becomes a network of disjunctive CUB-PTAs.

2. Normally, to synthesize parameter valuations on a network of several disjunctive CUB-PTAs with a single process, all disjunctive CUB-PTAs in the network must be synchronized (on-the-fly) to be a large single PTA. Unfortunately, this large PTA might cause some problems such as high memory consumption, time consuming etc.

   In our distribution scheme, to able distribute and avoid synthesizing on the large synchronized PTA, all disjunctive CUB-PTAs should not be synchronized at the same time, but some of them. In fact, each different split CUB-PTA in each disjunctive CUB-PTAs can be synchronized to have several partial synchronized PTAs.

   More precisely, assume each disjunctive CUB-PTA is a set of split CUB-PTAs. Then the number of partial synchronized PTAs is equal to the number of elements of the Cartesian product of all split CUB-PTA sets.

3. Since, a large synchronized PTA can be split into several partial synchronized PTAs. Then many processes can work on these partial synchronized PTAs.

Note that, the partial synchronized PTA will be called *setup* in Section 7.6.

Example 33. Suppose a network consists of 2 disjunctive CUB-PTAs in Fig. 7.2f and Fig. 7.3. The disjunctive CUB-PTA in Fig. 7.2f contains a set of 2 CUB-PTAs having initial constraints $p_1 \leq p_2$ and $p_1 > p_2$ called CUB-PTA1 and CUB-PTA2 respectively. The disjunctive CUB-PTA in Fig. 7.3 contains a set of 3 CUB-PTAs having initial constraints $p_1 \leq p_2 \wedge p_1 \leq p_3$, $p_1 \leq p_2 \wedge p_1 \geq p_3$ and

$p_1 \geq p_2$ called CUB-PTA1', CUB-PTA2' and CUB-PTA3' respectively. Then the Cartesian product of these two sets is a set including: (CUB-PTA1; CUB-PTA1'),..., (CUB-PTA2; CUB-PTA1'),..., (CUB-PTA2; CUB-PTA3'). Each element has 2 CUB-PTAs, which are synchronized to be a partial synchronized PTA. In the end, the master can distribute 6 partial synchronized PTAs to the workers.

7.6 Experiments

We implemented our algorithms in IMITATOR [AFKS12][4] with OCaml 4.02.3. The Parma Polyhedra Library (PPL v1.2) [BHZ08] is integrated inside the core of IMITATOR in order to solve mainly linear inequalities systems problems.

We consider various benchmarks to compare our techniques: protocols (CSMA/CD, Fischer [AHV93], RCP, WFAS), hardware circuits (And-Or, flip-flop), scheduling problems (Sched5), a networked automation system (simop) and various academic benchmarks.

7.6.1 Evaluation of the non-distributed version

Experiments were run on an Intel Core 2 Duo P8600 at 2.4 GHz with 4 GiB of memory, running Ubuntu 16.04 LTS 64-bit.

We compare three approaches: 1) A cycle detection synthesis without the non-Zenoness assumption (called synthCycle). The result may be an over-approximation of the actual result, as some of the parameters synthesized may yield only Zeno cycles. If synthCycle does not terminate, its result is an under-approximation of an over-approximation, therefore considered as potentially invalid; that is, there is no guarantee of correctness for the synthesized constraint.

2) Our CUB-detection (Algorithm 18) followed by synthesis (Algorithm 20): the result may be under-approximated, as only the valuations for which the PTA is CUB are considered. 3) Our CUB-transformation (CUBtrans in Algorithm 19) followed by synthesis (Algorithm 20) on the resulting disjunctive CUB-PTA. If the algorithm terminates, then the result is exact, otherwise it may be under-approximated. We use a modified version of the Tarjan algorithm for detecting SCCs.

We give from left to right in Table 7.1 the case study name and its number of clocks, parameters and locations. For synthCycle, we give the computation time (TO denotes a time-out at 3600 s), the constraint type ($\bot$, $\top$ or another constraint) and the validity of the result: if synthCycle terminates, the result is an over-approximation, otherwise it is potentially invalid. For CUBdetect (resp. CUBtrans) we give the detection (resp. transformation) time, the total time (including synthNZ), the result, and whether it is an under-approximation or an exact result. We also mention whether CUBdetect outputs that all, none or some valuations make the PTA CUB; and we give the number of locations in the

[4] Working version 2.9.2 (Build 2402 – distNZ/966d584). For experimental data including source and binary, see imitator.fr/static/NFM17

transformed disjunctive CUB-PTA output by CUBtrans. The percentage is used to compare the number of valuations (comparison obtained by discretization) output by the three algorithms, with CUBtrans as the basis (as the result is exact).

The toy benchmark CUBPTA1 is a good illustration: CUBtrans terminates after 0.073 s (and therefore its result is exact) with some constraint. CUBdetect is faster (0.015 s) but infers that only some valuations are CUB and analyzes only these valuations; the synthesized result is only 69 % of the expected result. In contrast, synthCycle is much faster (0.006 s) but obtains too many valuations (208 % of the expected result) as it infers many Zeno valuations.

Interpretation of the experiments Let us discuss the results in Table 7.1. First, synthCycle almost always outputs a possibly invalid result (neither an under- nor an over-approximation), which justifies the need for techniques handling non-Zeno assumptions. In only one case (CUBPTA1), it outputs a non-trivial over-approximation. In two cases, it happens to give an exact answer, as the over-approximation of $\bot$ necessarily means that $\bot$ is the exact result. In contrast, CUBtrans gives an exact result in five cases, a non-trivial under-approximation in two cases; the five remaining cases are a disappointing result in which $\bot$ is output as an under-approximation. By studying the model manually, we realized that some non-Zeno cycles actually exist for some valuations, but our synthesis algorithm was not able to derive them. Only in one of these cases (Sched5), synthCycle outputs a more interesting result than CUBtrans.

The transformation is relatively reasonable both in terms of added locations (in the worst case, there are 40 instead of 10 locations, hence four times more, for WFAS) and in terms of transformation time (the worst case is 1.2 s for Sched5). Our experiments do not allow us to fairly compare the time of synthCycle (without non-Zenoness) and synthNZ (with non-Zenoness assumption) as, without surprise due to the undecidability of synthesis, most analyses do not terminate. Only two benchmarks terminate for both algorithms, but are not significant (< 1 s).

Note that, flip-flop is a hardware circuit modeled using a bi-bounded inertial delay, and is therefore CUB for all valuations.

An interesting benchmark is WFAS, for which our transformation procedure terminates whereas synthCycle does not. Therefore, we get an exact result while the traditional procedure cannot produce any valuable output.

Conclusion As a conclusion, CUBdetect seems to be faster but less complete than CUBtrans. As for CUBtrans, its result is almost always more valuable than synthCycle, and therefore is the most interesting algorithm.

Model				synthCycle			CUBdetect					CUBtrans				
Name	**#** X	**#** P	**#** L	**time (s)**	**Result**	**Appr.**	**Detec time (s)**	**Total time (s)**	**CUB for**	**Result**	**Appr.**	**Trans time (s)**	**Total time (s)**	**#**L **CUB**	**Result**	**Appr.**
CSMA/CD	3	3	28	TO	$\top$	invalid?	0.013	0.013	$\bot$	-	-	0.300	TO	74	$\top$	exact
Fischer	2	4	13	TO	$\top$	invalid?	0.003	0.003	$\bot$	-	-	0.012	TO	20	$\top$	exact
RCP	6	5	48	TO	Some	invalid?	0.013	0.013	$\bot$	-	-	0.348	TO	71	$\bot$	under
WFAS	4	2	10	TO	Some 102%	invalid?	0.009	0.009	$\bot$	-	-	0.246	1848	40	Some 100%	exact
AndOr	4	4	27	TO	Some 166%	invalid?	0.012	0.012	$\bot$	-	-	0.059	TO	34	Some 100%	under
Flip-flop	5	2	52	0.058	$\bot$	exact	0.002	0.086	$\top$	$\bot$	exact	0.010	0.972	58	$\bot$	exact
Sched5	21	2	153	190	$\bot$	exact	0.051	0.051	$\bot$	-	-	1.180	TO	180	$\bot$	under
simop	8	2	46	TO	$\bot$	invalid?	0.012	0.012	$\bot$	-	-	0.219	TO	81	$\bot$	under
train-gate	5	9	11	TO	$\bot$	invalid?	0.000	TO	Some	$\bot$	under	0.059	TO	23	$\bot$	under
coffee	2	3	4	TO	Some 100%	invalid?	0.000	TO	Some	Some 100%	under	0.012	TO	10	Some 100%	under
CUBPTA1	1	3	2	0.006	$\top$ 208%	over	0.000	0.015	Some	Some 69%	under	0.006	0.073	6	Some 100%	exact
JLR13	2	2	2	TO	$\bot$	invalid?	0.000	TO	$\top$	$\bot$	under	0.000	TO	3	$\bot$	under

Table 7.1 – Experimental comparison of the three algorithms

7.6.2 Evaluation of the distributed version

Experiments were conducted on two Intel Xeon Silver 4114 at 2.2 GHz, and 96 GiB of memory, running under Ubuntu 16.04 LTS 64-bit.

In this section, we compare the performance of two algorithms, the CUB-transformation CUBtrans given in Algorithm 20, is re-named to synthNZ and its distributed version called distSynthNZ executed by the master node given in Algorithm 21.

The previous benchmarks from Table 7.1 are reused in this Table 7.2. Note that, due to the undecidability of non-Zeno emptiness in Section 7.2, our algorithms are not guaranteed to terminate.

From the left to right in Table 7.2, are the case study names, its number of clocks, parameters and locations. Because the number of split PTAs in the disjunctive CUB-PTA of original models is quite small for distSynthNZ, the number of parameters is increased to increase the number of CUB-PTAs in each disjunctive CUB-PTA. The next 2 columns indicate the number of disjunctive CUB-PTAs and the total number of CUB-PTAs are split in the benchmark. To compute a number of partial synchronized PTAs or setups in the next setup column, the number of CUB-PTAs in each disjunctive CUB-PTA is given in the parentheses (e. g. at CSMA/CD benchmark, the numbers in the parentheses (1-2-2) shows that CSMA/CD benchmark is a network of 3 disjunctive CUB-PTAs (or PTAs), the first one contains 1 CUB-PTA, the others contain 2 CUB-PTAs). For synthNZ, we give only the computation time for a single node. For distSynthNZ, we give the number of setups in the first column. Consider again the parentheses in the previous column, assume each number in the parentheses indicates a number of elements in a set. Then one can take the Cartesian product of these sets in the parentheses to have the number of setups (e. g. at CSMA/CD benchmark, the Cartesian product of the set (1-2-2) is 5). The next column indicates the maximum number of nodes can be used for a benchmark, computed by adding one to the number of setups (master and workers nodes). The last three columns are the computation time of distSynthNZ for 4, 6 and 8 nodes. In order to maximize the chances of our algorithm termination, a depth limit is set for all benchmarks, the algorithm will stop exploring at the depth of 200 and then returns an under-approximation constraint. We denote the algorithm reaches the depth limit of 200 by RD.

Interpretation of the experiments Let us discuss the result in Table 7.2. In our cases studies, RCP, Flip-flop, and train-gate are three large interesting cases since they are the only ones that our distributed algorithm can terminate and return exact results. Therefore, these cases will give a precise evaluation of the performance of distSynthNZ algorithm.

The table shows that distSynthNZ terminates 0.03, 1.47 and at least 11.45 times faster than synthNZ in RCP, Flip-flop and train-gate benchmark respectively. However, the disjunctive CUB-PTAs in RCP does not have many split CUB-PTAs, and requires only 2 setups or at most 3 nodes (2 for the workers and 1 for the master). Therefore, it makes sense that distSynthNZ does not gain

much performance in such situation. In the Flip-flop, the number of setups is increased to 24 and then the runtime of distSynthNZ is slightly improved in this case. Furthermore, in the train-gate benchmark, the number of setups is increased dramatically to 180. The state space explosion in this case is extremely huge and makes the synthNZ reaches time-out (3600s) while distSynthNZ needs 314s to terminate and return exact result. The 2 benchmarks at the bottom of the table can all terminate but they are quite small, and thus are not efficient with distSynthNZ. For the benchmarks reaching depth limit such as CSMA/CD, WFAS, coffee, their results are not so interesting since at the same depth, synthNZ and distSynthNZ give different numbers of explored states. Indeed, it is not entirely fair to compare the two algorithms with different state spaces. Finally, the remaining benchmarks suffer from the undecidable problem then cannot terminate.

Conclusion To summarize, our distributed algorithm distSynthNZ is useful for synthesizing non-Zeno runs on a large network of PTAs containing many constants and parameters. The experiment went as we expected, distSynthNZ dominates synthNZ in large benchmarks.

7.7 Conclusion

We proposed a technique to synthesize valuations for which there exists a non-Zeno infinite run in a PTA. By adding accepting states, this allows for parametric model checking with non-Zenoness assumption. Our techniques rely on a transformation to a disjunctive CUB-PTA (or in some cases on a simple detection of the valuation for which the PTA is already CUB), and then on a dedicated cycle synthesis algorithm. To cope with the state space explosion of large models, we proposed a distributed version of our synthesis algorithm on clusters. We implemented our techniques and its distributed version in IMITATOR and compared our algorithms on a set of benchmarks.

This is also the last contribution of the thesis, the next chapter will be dedicated to our final conclusion and future works.

Model						Single Node synthNZ		Multi-Node distSynthNZ					
Name	**# X**	**# P**	**# L**	**# DCUB -PTA**	**# CUB-PTA**	**Total t (s)**	**Appr.**	**# Setup**	**# Node Limit**	**4 Nodes Total t (s)**	**6 Nodes Total t (s)**	**8 Nodes Total t (s)**	**Appr.**
CSMA/CD	4	3	79	3	5 (1-2-2)	536.388 RD	under	4	5	991.670 RD	990.064 RD	-	under
Fischer	3	9	127	3	7 (1-5-1)	**TO**	under	5	6	**TO**	**TO**	-	under
RCP	7	7	89	5	6 (2-1-1-1-1)	948.819	exact	2	3	915.717	-	-	exact
WFAS	5	7	155	2	147 (7-7-3)	848.198 RD	under	147	148	644.748 RD	643.903 RD	645.675 RD	under
AndOr	4	14	71	4	4 (1-2-2-1)	**TO**	under	4	5	**TO**	**TO**	-	under
Flip-flop	6	15	165	5	11 (4-2-1-1-3)	686.555	exact	24	25	552.303	473.669	465.785	exact
Sched5	21	16	328	12	20 (5-1-2-1-2-1-2-1-2-1-1-1)	**TO**	under	80	81	**TO**	**TO**	**TO**	under
simop	9	7	254	5	48 (2-2-12-1-1)	**TO**	under	48	49	**TO**	**TO**	**TO**	under
train-gate	4	12	207	3	29 (18-10-1)	**TO**	under	180	181	441.322	353.966	314.320	exact
coffee	3	7	36	1	5	346.305 RD	under	5	6	473.347 RD	475.259 RD	-	under
CUBPTA1	3	9	101	1	80	0.983	exact	80	81	2.359	1.627	1.539	exact
JLR13	3	8	24	1	5	0.532	exact	5	6	1.106	1.123	-	exact

Table 7.2 – Experimental comparison of the non-distributed and distributed algorithms

Conclusion and perspectives

8.1 Summary of the thesis

The first contribution is in Chapter 3, where we designed distributed parameter synthesis algorithms to compute the cartography relying on the inverse method, and one of them, called dynamic domain decomposition Subdomain+H, is proved the most efficient for IM a parameter synthesis approach up-to-date.

Later on, in Chapter 4 we use a reachability preservation algorithm (PRP) instead of IM so as to obtain not a behavioral cartography but a simple "good/bad" partition with respect to a reachability property. We also bring the Subdomain+H distributed scheme to the PRP approach and again the distributed PRPC outperforms the monolithic bad-state reachability synthesis (e.g. [AHV93, JLR15]). It therefore makes sense that our Subdomain+H is highly reusable for other different purposes in the future than behavioral cartography BC based parameter synthesis approaches.

Zone inclusion is the well-known approach for the TA in term of state space reduction. It is also true with parametric zone inclusion for PTA, but in PTA computing the zone with parameters is more complicated and time-consuming than with TA. Furthermore, in Chapter 5 we showed that the parametric zone inclusion is not optimal and could explore unnecessary states, we called this problem the inefficient phenomenon. Thus, to alleviate the inefficient phenomenon, we optimize the parametric zone inclusion algorithm by proposing suitable exploration order strategies for it. We compared our new parametric ranking strategy RS (which comes from the idea of ranking strategy in [HT15]), and parametric priority strategy PRIOR with other traditional exploration orders for two main parameter synthesis problems. The result showed that the PRIOR strategy always gives better performance in most case studies. One weakness

of this algorithm is that it needs to store previously explored states to perform zone inclusion. Hence, to achieve better performance, this algorithm will be further changed to support parallel verification in a distributed setting with shared memory.

In Chapter 6, we proposed a new *nested depth-first search* (NDFS) algorithm and its variants for model-checking LTL properties of Parametric Timed Automata. This algorithm features several reduction mechanisms: subsumption, as in [LOD+13], early pruning, and a layered approach. It addresses both the problem of existence of an accepting cycle, and the synthesis of parameters that allow for a such a cycle in a collecting version. Our approach has the advantage of performing verification as early as possible, instead of fully exploring a branch of the parametric zone graph, which may be infinite. Experimental results show the efficiency of the layered approach. Subsumption and/or layering improves the efficiency of the algorithm, and also the chance of termination. We noted that the behaviour of subsumption with pure NDFS or with layered NDFS are quite complementary. Moreover, we have shown that such an approach can be used not only for Parametric Timed Automata but also for models featuring a progress measure which can be used to determine layers. Another nice feature is that the absence of cycles in the subsumed graph guarantees that no cycle exists in the PZG either, thus providing a quick answer when the formula holds.

As a first attempt to detect non-Zeno accepting run in PTA, in Chapter 7, we proposed an approach called CUB-PTA for non-Zeno parametric synthesis, which is derived from a clock upper-bound approach [WSW+15] for TA. Then, we proved that the problem of the valuations emptiness for which there exists at least non-Zeno accepting run is undecidable, and thus proposed a semi-algorithm for this CUB-PTA approach. Technically, we cannot synthesize valuations for which there exists at least one non-Zeno accepting run on the parametric zone graph of a PTA directly. To make it feasible, the key idea of this approach is to transform an input PTA into a disjunctive CUB-PTA, for which we can synthesize non-Zeno parameter valuations on its parametric zone graph. Additionally, to improve the performance of our approach on a large scale PTA containing many parameters and constants (resp. a network of PTAs), we distribute this semi-algorithm on a cluster by splitting a disjunctive CUB-PTA into multiple CUB-PTAs as in Section 7.5 (resp. a network of disjunctive CUB-PTAs into several partial synthesized PTAs as in Section 7.5.3), then we can distribute them easily on a cluster. Moreover, this is the first work of non-Zeno parameter synthesis for PTA: this said, there are still other techniques for non-Zeno model checking TA. They could be extended to PTAs, and could turn to be more efficient than our technique, and should therefore be investigated.

Finally, in order to valuate all approaches in the thesis, I have implemented most of the algorithms and distributed algorithms. The reader can find them with the benchmarks that we used in the IMITATOR repository [1].

[1] https://github.com/imitator-model-checker/imitator

8.2 Perspectives

In this section, we discuss some on-going and future works for parametric timed model checking and for the IMITATOR tool.

Distributed verification In addition to using powerful computing resources as in Chapter 3, our further aim is to design multi-core algorithms for parameter synthesis, in the line of [ELPvdP12, LOD+13] — then compare them and combine both approaches. Besides, we plan to extend swarm verification procedures proposed for Uppaal [ZNL16a, ZNL16b] and the novel parameter synthesis based on distributed CTL model checking of [BBD+16] would be of interest.

Finally, we would like to formally verify the master-worker communication scheme of Sections 3.3 and 3.4, so as to assure its reliability.

Parameter synthesis algorithms In the future, the work in Chapter 4 can be continued and improved by using the approach recently proposed to synthesize parameters using IC3 for reachability properties [CGMT13] which looks promising; it would be interesting to investigate a combination of that work with a PRP-like procedure.

Symbolic verification By studying subsumption abstraction in symbolic verification, we proposed some state exploration strategies in Chapter 5. There, we only gave the run-time of our algorithms. Then we should evaluate precisely the efficiency of our strategies by counting the number of inefficient phenomena for each strategy. Then we can compare the number of inefficient phenomena among strategies and this is very useful for strategy optimization purposes.

However, the idea of our algorithms come from the works in [HT15], in which the authors proposed the waiting strategy that we have not extended to the parametric case yet and it could serve as a basis for future parametric strategies. In addition, mitigating the cost of merging states used in Chapter 5 for our strategies other than LayerBFS is on our agenda, by selecting the right time to perform this expensive test.

As we mentioned in Section 2.5.4, the extrapolation or normalization for PTAs has not been studied yet. In order to deal with the termination of our algorithms, these techniques should be investigated.

Non-Zeno verification Our technique in Chapter 7 relying on CUB-PTAs extends the technique of CUB-TAs: this technique is shown in [WSW+15] to be the most efficient for performing non-Zeno model checking for TAs. However, for PTAs, other techniques (such as yet to be defined parametric extensions of strongly non-Zeno TAs [TYB05] or guessing the zone graph [HSW12]) could turn more efficient and should be investigated. Studying whether other techniques can be proposed is therefore on our agenda.

In addition, parametric stateful timed CSP (PSTCSP) [ALSD14] is a formalism for which the CUB assumption seems to be natively verified. Therefore,

studying non-Zeno parametric model checking for PSTCSP, as well as transforming PTAs into PSTCSP models, would be an interesting direction of research.

Finally, extending them to hybrid systems [SÁC+15] is also of high practical interest.

Decidability of subclasses The termination of a synthesis algorithm depends on its decidability problem. Although some techniques are proposed to increase the chance of termination and return a complete result, our procedures might not terminate. Therefore, studying the decidability of the underlying decision problem should be done for well-known subclasses of PTAs constraining the use of parameters (namely L/U-PTAs, L-PTAs and U-PTAs [HRSV02]) as well as for new semantic subclasses recently proposed and that benefit from decidability results (namely integer-point PTAs and reset-PTAs [ALR16a]).

Machine learning Machine learning, a prominent field related to artificial intelligence will be our new direction to increase the efficiency of parameter synthesis.

The very first work on machine learning for IMITATOR described in [LSGA17] looks quite promising. In this work, the authors use supervised learning to learn from non-parametric models for guessing a potential parameter constraint, which is then verified by calling IMITATOR. The result of [LSGA17] shows that machine learning is a promising direction for parameter synthesis.

In the future, it can be used to help us verifying large scale models and deal with the termination of our synthesis algorithms.

Bibliography

[ABB+16] Lacramioara Astefanoaei, Saddek Bensalem, Marius Bozga, Chih-Hong Cheng, and Harald Ruess. Compositional Parameter Synthesis. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 60–68, 2016. 13

[Abd12] Parosh Aziz Abdulla. Regular Model Checking. *STTT*, 14(2):109–118, 2012. 13

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *CAV*, Lecture Notes in Computer Science, pages 368–372. Springer, 2001. 26, 66

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 414–425. IEEE Computer Society, 1990. 34

[ACD+09] Étienne André, Thomas Chatain, Olivier De Smet, Laurent Fribourg, and Silvain Ruel. Synthèse de Contraintes Temporisées pour une Architecture d'Automatisation en Réseau. In Didier Lime and Olivier H. Roux, editors, *MSR*, Journal Européen des Systèmes Automatisés, pages 1049–1064. Hermès, November 2009. 66

[ACE14] Étienne André, Camille Coti, and Sami Evangelista. Distributed Behavioral Cartography of Timed Automata. In Jack Dongarra, Yutaka Ishikawa, and Hori Atsushi, editors, *21st European MPI Users' Group Meeting (EuroMPI/ASIA'14)*, pages 109–114. ACM, September 2014. 54, 55, 58, 81

[ACEF09] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An Inverse Method for Parametric Timed Automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009. 13, 27, 30, 40, 43, 119, 170

[ACN15] Étienne André, Camille Coti, and Hoang Gia Nguyen. Enhanced Distributed Behavioral Cartography of Parametric Timed Automata. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 319–335. Springer, 2015. 14

[AD90] Rajeev Alur and David L. Dill. Automata for Modeling Real-Time Systems. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990. 12, 13, 21

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 12, 13, 21, 74

[AD16] Parosh Aziz Abdulla and Giorgio Delzanno. Parameterized Verification. *STTT*, 18(5):469–473, 2016. 13

[ADD+11] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular Functions, Cost Register Automata, and Generalized Min-Cost problems. *CoRR*, abs/1111.0670, 2011. 13

[ADD+13] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular Functions and Cost Register Automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 13–22. IEEE Computer Society, 2013. 13

[AF10] Étienne André and Laurent Fribourg. Behavioral Cartography of Timed Automata. In Antonín Kučera and Igor Potapov, editors, *Proceedings of the 4th Workshop on Reachability Problems in Computational Models (RP'10)*, volume 6227 of *Lecture Notes in Computer Science*, pages 76–90. Springer, Aug 2010. 40, 41, 43, 171

[AFKS12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems. In *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36, 2012. 15, 32, 43, 54, 66, 81, 86, 94, 114, 141

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - A Tool for Modelling and Implementation of Embedded Systems. In *Tools and Algorithms for the Construction

and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, 2002. 26

[AFS13a] Étienne André, Laurent Fribourg, and Romain Soulat. Merge and Conquer: State Merging in Parametric Timed Automata. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2013. 86, 95

[AFS13b] Étienne André, Laurent Fribourg, and Jeremy Sproston. An Extension of the Inverse Method to Probabilistic Timed Automata. *Formal Methods in System Design*, 42(2):119–145, April 2013. 33

[AH89] Rajeev Alur and Thomas A. Henzinger. A Really Timed Automata. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 164–169. IEEE Computer Society, 1989. 34

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric Real-Time Reasoning. In *STOC*, pages 592–601. ACM, 1993. 13, 26, 27, 32, 33, 37, 42, 73, 74, 75, 84, 86, 94, 114, 120, 141, 147

[AKPP16] Étienne André, Michal Knapik, Wojciech Penczek, and Laure Petrucci. Controlling Actions and Time in Parametric Timed Automata. In *16th International Conference on Application of Concurrency to System Design, ACSD 2016, Torun, Poland, June 19-24, 2016*, pages 45–54, 2016. 33

[AL17] Étienne André and Didier Lime. Liveness in L/U-Parametric Timed Automata. In *17th International Conference on Application of Concurrency to System Design, ACSD 2017, Zaragoza, Spain, June 25-30, 2017*, pages 9–18. IEEE Computer Society, 2017. 36, 37, 38

[ALNS15] Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. Reachability Preservation Based Parameter Synthesis for Timed Automata. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015. 15, 72

[ALR16a] Étienne André, Didier Lime, and Olivier H. Roux. Decision Problems for Parametric Timed Automata. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 400–416. Springer, 2016. 37, 38, 39, 150

[ALR16b] Étienne André, Didier Lime, and Olivier H. Roux. On the Expressiveness of Parametric Timed Automata. In Martin Fränzle and Nicolas Markey, editors, *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec,*

QC, Canada, August 24-26, 2016, Proceedings, volume 9884 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2016. 37, 39

[ALSD14] Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Parameter Synthesis for Hierarchical Concurrent Real-Time Systems. *Real-Time Systems*, 50(5-6):620–679, 2014. 13, 149

[AM15] Étienne André and Nicolas Markey. Language Preservation Problems in Parametric Timed Automata. In *FORMATS*, volume 9268 of *Lecture Notes in Computer Science*, pages 27–43. Springer, 2015. 37, 39, 41, 120, 121

[And09] Étienne André. IMITATOR: A tool for synthesizing constraints on timing bounds of Timed Automata. In Martin Leucker and Carroll Morgan, editors, *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC'09)*, volume 5684 of *Lecture Notes in Computer Science*, pages 336–342, Kuala Lumpur, Malaysia, August 2009. Springer. 43

[And15] Étienne André. What's Decidable About Parametric Timed Automata? In *Formal Techniques for Safety-Critical Systems - Fourth International Workshop, FTSCS 2015, Paris, France, November 6-7, 2015. Revised Selected Papers*, pages 52–68, 2015. 13, 37

[And16] Étienne André. Parametric Deadlock-Freeness Checking Timed Automata. In *ICTAC*, volume 9965 of *Lecture Notes in Computer Science*, pages 469–478. Springer, 2016. 95

[ANP17] Étienne André, Hoang Gia Nguyen, and Laure Petrucci. Efficient Parameter Synthesis Using Optimized State Exploration Strategies. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*, pages 1–10. IEEE Computer Society, 2017. 15, 114

[ANPS17] Étienne André, Hoang Gia Nguyen, Laure Petrucci, and Jun Sun. Parametric Model Checking Timed Automata Under Non-Zenoness Assumption. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 35–51, 2017. 15

[AS11] Étienne André and Romain Soulat. Synthesis of Timing Parameters Satisfying Safety Properties. In *RP*, volume 6945 of *LNCS*, pages 31–44. Springer, 2011. 76

[AS13] Étienne André and Romain Soulat. *The Inverse Method*. ISTE Ltd and Wiley & Sons, 2013. 39, 40, 41, 86, 169, 170

[B+10] Blaise Barney et al. Introduction to Parallel Computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010. 44

[Bar] Blaise Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/, Last accessed on 2018-07-04. 45

[BBBC16] Peter Bezdek, Nikola Benes, Jiri Barnat, and Ivana Cerná. LTL Parameter Synthesis of Parametric Timed Automata. In Rocco De Nicola and eva Kühn, editors, *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, volume 9763 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2016. 87, 105

[BBD+16] Nikola Benes, Lubos Brim, Martin Demko, Samuel Pastva, and David Safránek. Parallel Smt-based Parameter Synthesis with Application to Piecewise Multi-Affine Systems. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 192–208, 2016. 55, 149

[BBLS15] Nikola Beneš, Peter Bezděk, Kim G. Larsen, and Jiří Srba. Language Emptiness of Continuous-Time Parametric Timed Automata. In *ICALP, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2015. 37

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998. 26

[BDR03] Véronique Bruyère, Emmanuel Dall'Olio, and Jean-François Raskin. Durations, Parametric Model-Checking in Timed Automata with Presburger Arithmetic. In Helmut Alt and Michel Habib, editors, *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, volume 2607 of *Lecture Notes in Computer Science*, pages 687–698. Springer, 2003. 14

[Bea03] Danièle Beauquier. On Probabilistic Timed Automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003. 13

[Beh05] Gerd Behrmann. Distributed Reachability Analysis in Timed Automata. *STTT*, 7(1):19–30, 2005. 86, 87

[BF01] Gerd Behrmann and Ansgar Fehnker. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2001. 87

[BFSV04] Giacomo Bucci, Andrea Fedeli, Luigi Sassoli, and Enrico Vicario. Timed State Space Analysis of Real-Time Preemptive Systems. *Transactions on Software Engineering*, 30(2):97–111, 2004. 83

[BG06] H. Bowman and R. Gómez. How to Stop Time Stopping. *Formal Aspects of Computing*, 18(4):459–493, 2006. 119, 121

[BHJL13] Béatrice Bérard, Serge Haddad, Aleksandra Jovanovic, and Didier Lime. Parametric interrupt Timed Automata. In *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, pages 59–69, 2013. 33

[BHV00] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing Timed Model Checking – How the Search Order Matters. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2000. 86, 87

[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. *Science of Computer Programming*, 72(1–2):3–21, 2008. 94, 114, 141

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. 11, 33, 34

[BL09] Laura Bozzelli and Salvatore La Torre. Decision Problems for Lower/Upper Bound Parametric Timed Automata. *Formal Methods in System Design*, 35(2):121–151, 2009. 34, 38, 39, 170

[BLN03] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *CAV*, pages 122–125. Springer, 2003. 26

[BLP03] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. To Store or Not to Store. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer, 2003. 87

[BO14] Daniel Bundala and Joël Ouaknine. Advances in Parametric Real-Time Reasoning. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2014. 37

[BOS02] Víctor A. Braberman, Alfredo Olivero, and Fernando Schapachnik. ZEUS: A Distributed Timed Model-Checker Based on KRONOS.

Electronic Notes in Theoretical Computer Science, 68(4):503–522, 2002. 87

[BY03] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. 32, 39, 85, 87

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. 13

[CC04] Robert Clarisó and Jordi Cortadella. Verification of Timed Circuits with Symbolic Delays. In Masaharu Imai, editor, *ASP-DAC*, pages 628–633, Piscataway, NJ, USA, 2004. IEEE Computer Society. 66

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, 1981. 34

[CETFX09] Rémy Chevallier, Emmanuelle Encrenaz-Tiphene, Laurent Fribourg, and Weiwen Xu. Timed Verification of the Generic Architecture of a Memory Circuit Using Parametric Timed Automata. *Form. Methods Syst. Des.*, 34(1), February 2009. 33, 83

[CGMT13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter Synthesis with IC3. In *FMCAD*, pages 165–168. IEEE, 2013. 54, 149

[CPR08] Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *RTSS*, pages 80–89. IEEE Computer Society, 2008. 33

[CVWY92] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992. 43

[Der00] John Derrick. Concurrent and Real-Time Systems: The CSP Approach, Steve Schneider, Wiley, 2000 (Book Review). *Softw. Test., Verif. Reliab.*, 10(3):195, 2000. 12, 13

[DHQ+08] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed Automata Patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008. 127

[DLL+12] A.E. Dalsgaard, A.W. Laarman, K.G. Larsen, M.C. Olesen, and J.C. van de Pol. Multi-Core Reachability for Timed Automata. In *FORMATS*, LNCS 7595, 2012. 32

[Doy07] Laurent Doyen. Robust Parametric Reachability for Timed Automata. *Inf. Process. Lett.*, 102(5):208–213, 2007. 37

[DSZ10] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized Verification of Ad Hoc Networks. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2010. 13

[DT98] Conrado Daws and Stavros Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 1998. 31, 87

[DWDR05] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Almost ASAP Semantics: From Timed Models to Timed Implementations. *Formal Aspects of Computing*, 17(3):319–341, 2005. 54

[EH86] E. Allen Emerson and Joseph Y. Halpern. ”Sometimes” and ”Not Never” Revisited: on Branching Versus Linear Time Temporal Logic. *J. ACM*, 33(1):151–178, 1986. 34

[EK10] Sami Evangelista and Lars Michael Kristensen. Search-Order Independent State Caching. *Transactions on Petri Nets and Other Models of Concurrency*, 4:21–41, 2010. 87

[EK14] Sami Evangelista and Lars Michael Kristensen. A Sweep-Line Method for Büchi Automata-based Model Checking. *Fundam. Inform.*, 131(1):27–53, 2014. 105, 112

[ELPvdP12] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol. Improved Multi-Core Nested Depth-First Search. In *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012. 31, 54, 149

[FJ13] Léa Fanchon and Florent Jacquemard. Formal Timing Analysis of Mixed Music Scores. In *Proceedings of the 39th International Computer Music Conference, ICMC 2013, Perth, Australia, August 12-16, 2013*, 2013. 33

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1996. 43

[FJK08] G. Frehse, S.K. Jha, and B.H. Krogh. A Counterexample-Guided Approach to Parameter Synthesis for Linear Hybrid Automata. In *HSCC'08*, volume 4981 of *LNCS*, pages 187–200. Springer, 2008. 39

[FSLM12] Laurent Fribourg, Romain Soulat, David Lesens, and Pierre Moro. Robustness Analysis for Scheduling Problems Using the Inverse Method. In *19th International Symposium on Temporal Representation and Reasoning, TIME 2012, Leicester, United Kingdom, September 12-14, 2012*, pages 73–80, 2012. 33

[GB07] R. Gómez and H. Bowman. Efficient Detection of Zeno Runs in Timed Automata. In *FORMATS*, volume 4763 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2007. 119

[Geo14] Samuel R. J. George. *Design, Formalization and Realization of Harmonic Box Coordination Language : An Externally Timed Specification Substrate for Arbitrarily Reliable Distributed Systems*. PhD thesis, University of Bristol, UK, 2014. 13

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004. 45

[GHP92] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-Space Caching Revisited. In *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191. Springer, 1992. 87

[Har05] Jon D Harrop. *OCaml for Scientists*. Flying Frog Consultancy, 2005. 43

[HBE+10] Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou, and Sylvain Peyronnet. Three High Performance Architectures in the Parallel APMC Boat. In *PMDC*. IEEE, 2010. 54

[HBH+99] Anthony Hall, Jonathan P. Bowen, Michael G. Hinchey, Jeannette M. Wing, and C. A. R. Hoare. Formal Methods. In *High-Integrity System Specification and Design*, Formal Approaches to

Computing and Information Technology (FACIT), pages 127–230. Springer London, 1999. 10

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, 1996. 21

[HH94] Thomas A. Henzinger and Pei-Hsin Ho. HYTECH: The Cornell HYbrid TECHnology Tool. In *Hybrid Systems II*, pages 265–293, 1994. 33

[HHW97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *STTT*, 1(1-2):110–122, 1997. 33

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation - International Edition (2. ed)*. Addison-Wesley, 2003. 165, 166, 167

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. 12

[HRSV02] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear Parametric Model Checking of Timed Automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002. 27, 32, 38, 86, 87, 119, 131, 150, 174

[HS10] Frédéric Herbreteau and B. Srivathsan. Efficient On-the-Fly Emptiness Check for Timed Büchi Automata. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2010. 14

[HSTW16] Frédéric Herbreteau, B. Srivathsan, Thanh-Tung Tran, and Igor Walukiewicz. Why Liveness for Timed Automata Is Hard, and What We Can Do About It. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, pages 48:1–48:14, 2016. 105

[HSW12] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Efficient Emptiness Check for Timed Büchi Automata. *Formal Methods in System Design*, 40(2):122–146, 2012. 119, 149

[HT15] Frédéric Herbreteau and Thanh-Tung Tran. Improving Search Order for Reachability Testing in Timed Automata. In *FORMATS*, volume 9268 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2015. 86, 87, 89, 91, 147, 149

[JLR15] Aleksandra Jovanović, Didier Lime, and Olivier H. Roux. Integer Parameter Synthesis for Timed Automata. *TSE*, 41(5):445–461, 2015. 5, 13, 29, 30, 32, 33, 35, 37, 38, 42, 51, 54, 73, 74, 75, 82, 83, 84, 86, 94, 119, 147, 173

[KMH01] Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping Temporal Planning Constraints into Timed Automata. In *TIME*, pages 21–27. IEEE Computer Society, 2001. 26

[KMP15] Michal Knapik, Artur Meski, and Wojciech Penczek. Action Synthesis for Branching Time Logic: Theory and Applications. *ACM Trans. Embedded Comput. Syst.*, 14(4):64:1–64:23, 2015. 13, 33

[KNSS99] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic Verification of Real-Time Systems with Discrete Probability Distributions. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings*, volume 1601 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 1999. 13

[Koy90] Ron Koymans. Specifying Real-Time Properties with Metric Timed Automata. *Real-Time Systems*, 2(4):255–299, 1990. 34

[KP12] Michal Knapik and Wojciech Penczek. Bounded Model Checking for Parametric Timed Automata. *Trans. Petri Nets and Other Models of Concurrency*, 5:141–159, 2012. 13, 32, 94, 114

[KT11] Temesghen Kahsai and Cesare Tinelli. PKind: A Parallel k-induction Based Model Checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011. 54

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. 34

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995. 13

[LOD+13] Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim Guldstrand Larsen, and Jaco Van De Pol. Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013. 54, 87, 105, 106, 107, 108, 148, 149

[Lon93] David Esley Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1993. UMI Order No. GAX94-02579. 44

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997. 26

[LRST09] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 54–57, 2009. 26

[LSGA17] Jiaying Li, Jun Sun, Bo Gao, and Étienne André. Classification-Based Parameter Synthesis for Parametric Timed Automata. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering - 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*, volume 10610 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2017. 150

[Mar11] Nicolas Markey. Robustness in Real-Time Systems. In *SIES*, pages 28–34. IEEE Computer Society Press, 2011. 26, 40, 54

[Mer74] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems.* PhD thesis, University of California, Irvine, 1974. AAI7511026. 12, 13

[Mil00] Joseph S. Miller. Decidability and Complexity Results for Timed Automata and Semi-Linear Hybrid Automata. In Nancy A. Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000, Proceedings*, volume 1790 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000. 37

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967. 37, 120

[MMH13] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media Inc., 2013. 43

[NPvdP18] Hoang Gia Nguyen, Laure Petrucci, and Jaco van de Pol. Layered and Collecting NDFS with Subsumption for Parametric Timed Automata. In *23nd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*. IEEE Computer Society, 2018. 15

[OW02] Joël Ouaknine and James Worrell. Timed CSP = Closed Timed Safety Automata. *Electr. Notes Theor. Comput. Sci.*, 68(2):142–159, 2002. 12, 13

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962. 12

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 1977. 34

[SÁC+15] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Goran Frehse, Sriram Sankaranarayanan, and Stefan Kowalewski. Current Challenges in the Verification of Hybrid Systems. In *CyPhy*, volume 9361 of *Lecture Notes in Computer Science*, pages 8–24. Springer, 2015. 150

[SE05] S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *TACAS*, LNCS 3440, pages 174–190. Springer, 2005. 108

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009. 26

[TLR09] Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric Model-Checking of Stopwatch Petri Nets. *J. UCS*, 15(17):3273–3304, 2009. 13, 33

[Tri99] Stavros Tripakis. Verifying Progress in Timed Systems. In *AMAST*, pages 299–314, 1999. 14, 119, 121, 131

[TYB05] Stavros Tripakis, Sergio Yovine, and Ahmed Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005. 14, 119, 149

[Wan02] F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In *Formal Techniques for Networked and Distributed Systems*, pages 235–250. Springer, 2002. 26

[WR88] W. M. Wonham and P. J. Ramadge. Modular Supervisory Control of Discrete-Event Systems. *MCSS*, 1(1):13–30, 1988. 13

[WSW+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. *IEEE Transactions on Software Engineering*, 41(1):3–18, 2015. 14, 119, 121, 122, 123, 127, 128, 131, 148, 149

[ZNL16a] Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. Distributed Algorithms for Time Optimal Reachability Analysis. In Martin Fränzle and Nicolas Markey, editors, *Formal Modeling and Analysis*

of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings, volume 9884 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2016. 55, 149

[ZNL16b] Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. Time Optimal Reachability Analysis Using Swarm Verification. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1634–1640. ACM, 2016. 55, 149

Appendix

7

Contents

A.1 Decidability **165**
A.1.1 Turing machine 165
A.1.2 Halting problem 166
A.1.3 Decidable and undecidable problems 167
A.1.4 Reducibility 167
A.2 Two-counter machine **168**

A.1 Decidability

Decidability studies problems that computer has ability to solve. A certain problem is said to be decidable if it can be solved by a computer.

A.1.1 Turing machine

Recall in the literature, e. g. [HMU03], the Turing machine is a simple computer-like model or a finite automaton, has a single infinite tape on which it can read and write data.

Definition A.1.1. A Turing machine (TM for short) is a tuple:
$\mathsf{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where:

- Q: The finite set of states of the finite control.
- Σ: The finite set of input symbols.
- Γ: The complete set of tape symbols; Σ is always a subset of Γ.
- q_0: The initial state and $q_0 \in Q$.
- B: The blank symbol, $B \in \Gamma$ but $B \notin \Sigma$; it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.
- F: The set of final states, $F \in Q$
- δ: The transition function. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$ is a triple (p, Y, D) where:
  1. p: is the next state in Q.
  2. Y: is the symbol in Γ, written in the cell being scanned, replacing whatever symbol was there.
  3. D: is a direction either L or R, standing for "left" or "right", respectively, and telling us the direction in which the head moves.

A.1.2 Halting problem

In computability theory, the halting problem is a decision problem. It asks "given a computer program and an input, will the program terminate or will it run forever?". In general, a computer program with its input can be modeled by a TM with an infinite tap. A TM halts if it enters a state q and then scanning a tape symbol X, and there is no more transition $\delta(q, X)$ to move to other states.

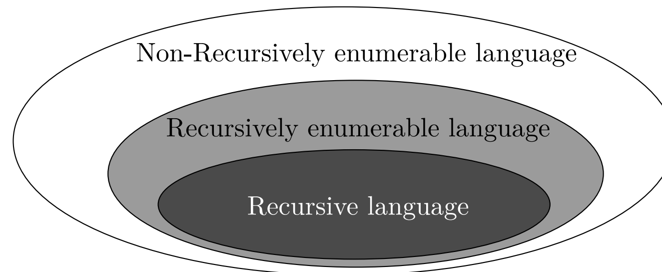
Languages accepted by the Turing machine [HMU03]

Recursively enumerable language : A language is recursively enumerable if some Turing machine accepts it. The Turing machine cannot be halt on this language and loop forever. And there is an enumeration procedure for it.

Recursive language : A language is recursive if some Turing machine accepts it and halts on any input string. And iff there is an enumeration procedure for it.

Non-Recursively enumerable language : A language is not accepted by any Turning machine.

In other words, recursive language ⊂ recursively enumerable language ⊂ non-recursively enumerable language:



A.1.3 Decidable and undecidable problems

Recall again in [HMU03], given a TM (algorithm), a language (problem) is called decidable if it is a recursive language and it is called undecidable if it is not a recursive language.

Decidable problems If there is an algorithm to solve a given problem exists, that always finishes and produces an answer, then we say the problem is decidable. In other words, there is a TM will halt on its input eventually (whether it accepts or not). Hence, TMs that always halt are a good model of an algorithm.

Undecidable problems Otherwise, if there does not exist an algorithm to solve a given problem, then we say the problem is undecidable. In other words, there is no TM accepting its input or it will run forever and fails to halt on its input.

A.1.4 Reducibility

Sometimes a certain problem cannot be solved directly, then a problem **A** will be reduced to problem **B**, and if we can solve the problem **B** then we can solve problem **A**.

It is the same with the decidable problem, problem **A** is reduced to problem **B**. If **B** is decidable then **A** is decidable, on the contrary if A is undecidable then **B** is undecidable.

A.2 Two-counter machine

Definition A.2.1. A two-counter machine M is a finite state machine with two integer-valued counters $\mathtt{c}_1, \mathtt{c}_2$. Two different instructions (presented for $\mathtt{c}_1$ and identical for $\mathtt{c}_2$) are considered: for $\mathtt{c}_2$ are similar) *i*) when in state $\mathtt{q}_i$, increment $\mathtt{c}_1$ and go to $\mathtt{q}_j$; *ii*) when in state $\mathtt{q}_i$, if $\mathtt{c}_1 = 0$ go to $\mathtt{q}_k$, otherwise decrement $\mathtt{c}_1$ and go to $\mathtt{q}_j$.

We assume w.l.o.g. that the machine halts iff it reaches a special state $\mathtt{q}_{\mathrm{halt}}$.

Appendix B

Distributed verification of parametric real-time systems

Contents

B.1 Existing algorithms **169**

B.1.1 The Inverse Method algorithm 169

B.1.2 The Behavioral Cartography algorithm 171

B.2 Master-worker point distribution algorithms **171**

B.2.1 Sequential point distribution: initialization algorithm 171

B.2.2 Random point distribution: initialization algorithm . 171

B.2.3 Shuffle point distribution: algorithms 172

B.1 Existing algorithms

B.1.1 The Inverse Method algorithm

We recall here IM from [AS13].

Given a parameter valuation v, a state (l, C) is said to be v-compatible if $v \models C$. We extend Succ to sets of states as follows: given a set S of states, $\mathsf{Succ}(S) = \{s' \mid \exists s \in S \text{ s.t. } s' \in \mathsf{Succ}(s)\}$. Given a set S of symbolic states, we denote by $\mathsf{Succ}^j(S)$ the set of states reachable from S in exactly j steps, i. e. the composition of j times Succ.

Here, we consider PTA extended with an initial parameter domain (as considered in, e. g. [AS13, ACEF09, BL09]). That is, these PTA have their possible parameter valuations restricted to belong to the set defined by a parameter constraint K. When clear from the context, given a PTA $\mathcal{A}$ and a constraint K, we denote by $\mathcal{A}(K)$ the PTA initially constrained by $\mathcal{A}$. (This can be simulated using an initial gadget that will ensure this constraint over the parameters before the actual initial location of the PTA.)

We use notation $\mathsf{Succ}_{\mathcal{A}(K)}$ to denote that the Succ operation is applied to PTA $\mathcal{A}(K)$.

Algorithm 23: Inverse Method $\mathsf{IM}(\mathcal{A}, v)$

---

input : PTA $\mathcal{A}$, parameter valuation v
output : Constraint K over the parameters

```
1  i ← 0 ;  K_c ← ⊤ ;  S_new ← {s_0} ;  S ← {}
2  while true do
3      while there are v-incompatible states in S_new do
4          Select a v-incompatible state (l, C) of S_new
5          Select a v-incompatible J in C↓P
6          K_c ← K_c ∧ ¬J
7          S ← ⋃_{j=0}^{i-1} Succ^j_{A(K_c)}({s_0})
8          S_new ← Succ_{A(K_c)}(S)
9      if S_new ⊆ S then
10         return K ← ⋂_{(l,C)∈S} C↓P
11     i ← i + 1 ;  S ← S ∪ S_new ;  S_new ← Succ_{A(K_c)}(S)
```

IM [AS13, ACEF09] is a breadth-first algorithm (given in Algorithm 23), that maintains an integer i (which corresponds to the exploration depth), the current constraint K_c (initially set to $\top$, i. e. the parameter constraint corresponding to all parameter valuations), the set S_{new} of states computed at the latest iteration, and the set S of states computed at all previous iterations. IM iteratively explores states and refines the constraint K_c: when a v-incompatible state (l, C) is met (line 4), then a v-incompatible inequality is selected within $C{\downarrow_P}$ (line 5), and added to K_c (line 8). When a fixpoint is reached, i. e. when no more new state is generated (line 9), then the intersection of the projection onto P of all reachable states is returned (line 10).

The inverse method can be characterized as follows.

Theorem B.1.1 ([AS13])**.** *Let $\mathcal{A}$ be a PTA and v be a parameter valuation. Assume $\mathsf{IM}(\mathcal{A}, v)$ terminates with result K. Then*

1. *$v \models K$, and*
2. *for all $v' \models K$, the trace sets of $\mathcal{A}[v]$ and $\mathcal{A}[v']$ are the same.*

B.1.2 The Behavioral Cartography algorithm

We recall the original non-distributed behavioral cartography algorithm from [AF10] in Algorithm 24. We extend the $\models$ notation as follows: given a set T of tiles, we write $v \models T$ if there exists some K in T such that $v \models K$.

Algorithm 24: Behavioral Cartography $\mathsf{BC}(\mathcal{A}, D)$

```
   input    : PTA 𝒜, point v
   output   : Set of tiles T
1  T ← ∅
2  foreach integer point v ∈ D do
3  ⌊ if v ⊭ T then T ← T ∪ {IM(𝒜, v)};
4  return T
```

B.2 Master-worker point distribution algorithms

B.2.1 Sequential point distribution: initialization algorithm

We give the $\mathsf{Sequential}.\mathit{initialize}()$ function in Algorithm 25.

Algorithm 25: $\mathsf{Sequential}.\mathit{initialize}()$

```
   variables : Point v_prev
1  v_prev ← ⊥
```

B.2.2 Random point distribution: initialization algorithm

We give the $\mathsf{Random}.\mathit{initialize}()$ function in Algorithm 26.

Algorithm 26: $\mathsf{Random}.\mathit{initialize}()$

```
   variables : Point v_prev, flag seqPhase
1  seqPhase ← false
2  v_prev ← ⊥
```

B.2.3 Shuffle point distribution: algorithms

Algorithm 27: Shuffle.$\mathit{initialize}()$

---

variables : List of points $\mathit{allPoints}$

1 $\mathit{allPoints} \leftarrow \mathit{shuffle}(\mathit{allIntegers}(D))$

Algorithm 28: Shuffle.$\mathit{choosePoint}()$

---

variables : List of points $\mathit{allPoints}$
output : Point v

1 $v \leftarrow \mathit{pop}(\mathit{allPoints})$
2 **while** *v is covered by a tile* **do**
3     $v \leftarrow \mathit{pop}(\mathit{allPoints})$
4 **return** v

APPENDIX C

Reachability preservation based parameter synthesis for timed automata

Contents

C.1 Reachability synthesis algorithm 173

C.2 Proof of Proposition 4.2.3 174

C.1 Reachability synthesis algorithm

We first recall in Algorithm 29 the semi-algorithm EFsynth in [JLR15]. If EFsynth terminates, it synthesizes all parameter valuations for which a set of target location G is reachable, and it is also a correct solution to the EF-synthesis problem (see the proof in [JLR15]).

Technically, the semi-algorithm EFsynth below is a recursive function which explores all possible states on the symbolic reachability tree, and collects all parametric constraints associated with the target locations G . The set S is used to remember the visited states. The initial call to EFsynth is set with EFsynth$(\mathcal{A}; \mathbf{s}_0; G; \emptyset)$. If the state is a target sate, the projection of its constraint onto the parameters is returned (line 1). Otherwise, the algorithm returns the union over all outgoing edges of the algorithm recursively applied to its successors

via these edges (line 6).

Algorithm 29: Reachability synthesis EFsynth($\mathcal{A}, s, G, S$)

input : A PTA $\mathcal{A}$, a symbolic state $s = (l, C)$, a set of target locations G, a set S of passed states on the current path
output : Constraint K over the parameters

```
1 if l ∈ G then K ← C↓P ;
2 else
3     K ← ⊥;
4     if s ∉ S then
5         for each outgoing edge e from l in A do
6             K ← K ∪ EFsynth(A, Succ(s, e), G, S ∪ {s});
7 return K
```

C.2 Proof of Proposition 4.2.3

We first recall below two results from [HRSV02], that relate runs in TA and PTA.

Proposition C.2.1 ([HRSV02, Proposition 3.17])**.** *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. For any symbolic run of $\mathcal{A}$ reaching a state (l, C), there exists an equivalent concrete run in $\mathcal{A}[v]$ reaching a state (l, w), with $<w|v> \models C$.*

Proposition C.2.2 ([HRSV02, Proposition 3.18])**.** *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. For any concrete run of $\mathcal{A}[v]$ reaching a state (l, w), there exists an equivalent symbolic run in $\mathcal{A}$ reaching a state (l, C) such that $<w|v> \models C$.*

We can now prove Proposition 4.2.3 below.

Proposition C.2.3. *Let $\mathcal{A}$ be a PTA, and v a parameter valuation. Suppose PRP($\mathcal{A}, v$) terminates with result K. Then, $v \models K$ and, for all $v' \models K$:*

- *if l_{bad} is reachable in $\mathcal{A}[v]$, then l_{bad} is reachable in $\mathcal{A}[v']$;*
- *if l_{bad} is unreachable in $\mathcal{A}[v]$, then every trace of $\mathcal{A}[v']$ is a trace of $\mathcal{A}[v]$.*

Proof. Consider the value of *Bad* at the end of PRP.

- First, suppose $Bad = \texttt{true}$. From Algorithm 7, some bad states have been met, and the output is K_{bad}. Let us first note that $v \models K_{bad}$: by construction, K_{bad} is made of the disjunction of constraints associated with

states of S_{new} that are necessarily v-compatible – otherwise they would have been discarded (lines 3–7).

Now, let $v' \models K_{bad}$. K_{bad} is made of a disjunction of constraints of the form $K_{bad} = K^1_{bad} \vee \cdots \vee K^n_{bad}$ for some n. Since $v' \models K_{bad}$, there exists at least one K^j_{bad} with $1 \leq j \leq n$ such that $v' \models K^j_{bad}$. This constraint K^j_{bad} has been added in Algorithm 7 when a run reaching some (l_{bad}, C^j) has been met (line 9), with $K^j_{bad} = C^j{\downarrow_P}$. Since $v' \models K^j_{bad}$, then $v' \models C^j$ hence there exists w' such that $<w'|v'> \models C^j$. Hence, from Proposition C.2.1, there exists an equivalent run in $\mathcal{A}[v']$, and therefore l_{bad} is reachable in $\mathcal{A}[v']$.

Using the same reasoning with v together with the fact that $v \models K_{bad}$ gives that l_{bad} is also reachable in $\mathcal{A}[v]$.

- Conversely, suppose $Bad =$ `false`. From Algorithm 7, no bad state has been met, and the output is K_{good}.

  Let us first note that $v \models K_{good}$: by construction of K_{good}, K_{good} is made of the intersection of the negation of v-incompatible inequalities.

  Let us now show that every trace of $\mathcal{A}[v']$ is a trace of $\mathcal{A}[v]$. Let $v' \models K_{good}$, and consider a run of $\mathcal{A}[v']$ reaching (l, w'). From Proposition C.2.2, there exists an equivalent run in $\mathcal{A}$ reaching a state (l, C) with $<w'|v'> \models C$.

  1. First, assume $v \models C{\downarrow_P}$: by definition, there exists w such that $<w|v> \models C$. Hence, from Proposition C.2.1, there exists an equivalent run in $\mathcal{A}[v]$ reaching (l, w), which gives the result.
  2. Conversely, assume $v \not\models C{\downarrow_P}$: this situation cannot happen due to the removal of v-incompatible states in Algorithm 7.

  Now, we shall show that l_{bad} is unreachable in $\mathcal{A}[v]$. Suppose l_{bad} is reachable in $\mathcal{A}[v]$, i. e., there exists a run of $\mathcal{A}[v]$ reaching (l_{bad}, w), for some w. From Proposition C.2.2, there exists an equivalent run in $\mathcal{A}$ reaching a state (l_{bad}, C) with $<w|v> \models C$. This cannot happen since C is by definition v-compatible, and no bad state was met in Algorithm 7.

From the two items above, we have that l_{bad} is reachable in $\mathcal{A}[v]$ iff $Bad =$ `true`. Hence, if l_{bad} is reachable in $\mathcal{A}[v]$, then l_{bad} is reachable in $\mathcal{A}[v']$; and conversely if l_{bad} is unreachable in $\mathcal{A}[v]$, then every trace of $\mathcal{A}[v']$ is a trace of $\mathcal{A}[v]$. □