ENSC-2010 n°257

THÈSE

présentée à l'École Normale Supérieure de Cachan

en vue de l'obtention du grade de

Docteur de l'École Normale Supérieure de Cachan

par Étienne ANDRÉ

Spécialité informatique

An Inverse Method for the Synthesis of Timing Parameters in Concurrent Systems

Soutenue le 8 décembre 2010 devant un jury composé de

Eugene ASARIN	Président du jury
Bernard BERTHOMIEU	Rapporteur
Franck CASSEZ	Rapporteur
Emmanuelle ENCRENAZ-TIPHENE	Co-directrice de thèse
Laurent FRIBOURG	Co-directeur de thèse
Marta KWIATKOWSKA	Examinatrice
Kim G. LARSEN	Examinateur

Abstract

This thesis proposes a novel approach for the synthesis of delays for timed systems. When verifying a real-time system, e.g., a hardware device or a communication protocol, it is important to check that not only the functional but also the timed behavior is correct. This correctness depends on the values of the delays of internal operations and of the environment.

Formal verification methods guarantee the correctness of a timed system for a given set of delays, but do not give information for other values of the delays. Checking the correctness of for various values of those delays can be difficult and time consuming. It is thus interesting to consider that these delays are parameters. The problem then consists in synthesizing "good values" for those parameters, i.e., values for which the system is guaranteed to behave well.

We are here interested in the synthesis of parameters in the framework of timed automata, a model for verifying real-time systems. Our approach relies on the following inverse method: given a reference valuation of the parameters, we synthesize a constraint on the parameters, guaranteeing the same time-abstract linear behavior as for the reference valuation. This gives a criterion of robustness to the system. By iterating this inverse method on various points of a bounded parameter domain, we are then able to partition the parametric space into good and bad zones, with respect to a given property one wants to verify. This gives a behavioral cartography of the system.

This method extended to probabilistic systems allows to preserve minimum and maximum probabilities of reachability properties. We also present variants of the inverse method for directed weighted graphs and Markov Decision Processes. Several prototypes have been implemented; in particular, IMITATOR II implements the inverse method and the cartography for timed automata. It allowed us to synthesize parameter values for several case studies such as an abstract model of a memory circuit sold by the chipset manufacturer ST-Microelectronics, and various communication protocols.

Keywords: verification, model checking, timed systems, parametric timed automata, synthesis of parameters, hardware verification, probabilistic timed automata, randomized protocols.

Résumé

Cette thèse propose une nouvelle approche pour la synthèse de valeurs temporelles dans les systèmes temporisés. Lorsque l'on vérifie un système temps-réel, comme un circuit ou un protocole de communication, il est important de vérifier non seulement l'aspect fonctionnel, mais également temporisé. La correction du système dépend de valeurs temporelles internes et de l'environnement.

Les méthodes formelles de vérification garantissent la correction d'un système temporisé pour un ensemble de valeurs temporelles, mais ne donnent pas d'information pour d'autres valeurs. Vérifier la correction d'un système pour de nombreuses valeurs peut s'avérer long et difficile. Il est alors intéressant de les considérer comme paramètres. Le problème consiste alors à synthétiser des valeurs de ces paramètres pour lesquelles le système est correct.

Nous nous intéressons ici à la synthèse de paramètres dans le cadre des automates temporisés. Notre approche est basée sur la méthode inverse suivante : à partir d'une instance de référence des paramètres, nous synthétisons une contrainte sur les paramètres, garantissant le même comportement que pour l'instance de référence, abstraction faite du temps. Il en résulte un critère de robustesse pour le système. En itérant cette méthode sur des points dans un domaine paramétrique borné, nous sommes alors à même de partitionner l'espace des paramètres en bonnes et mauvaises zones par rapport à une propriété à vérifier. Ceci nous donne une cartographie comportementale du système.

Cette méthode s'étend aisément aux systèmes probabilistes. Nous présentons également des variantes de la méthode inverse pour les graphes orientés valués et les processus de décision markoviens. Parmi les prototypes implémentés, IMITATOR II implémente la méthode inverse et la cartographie pour les automates temporisés. Ce prototype nous a permis de synthétiser de bonnes valeurs pour les paramètres temporels de plusieurs études de cas, dont un modèle abstrait d'une mémoire commercialisée par le fabricant de puces ST-Microelectronics, ainsi que plusieurs protocoles de communication.

Mots-clés : vérification, model-checking, systèmes temporisés, automates temporisés paramétrés, synthèse de paramètres, vérification de circuits, automates temporisés probabilistes, protocoles de communication.

Remerciements

Je remercie en premier lieu mes directeurs de thèse Emmanuelle ENCRENAZ et Laurent FRIBOURG pour les pistes de recherche qu'ils ont su me conseiller au cours de ces trois années, pour leurs conseils avisés, pour nos discussions parfois contradictoires et toujours bénéfiques.

Je tiens également à remercier Bernard BERTHOMIEU et Franck CASSEZ pour m'avoir fait l'honneur de relire ma thèse, et y avoir apporté de nombreux commentaires et suggestions d'améliorations constructifs. I would also like to thank Eugene ASARIN, Marta KWIATKOWSKA and Kim G. LARSEN to have done me the great honor of accepting to take part to my jury.

Un grand merci également à plusieurs personnes qui m'ont, d'une manière ou d'une autre, encouragé à entreprendre une thèse : Patrice QUINTON en premier lieu pour ses conseils très judicieux, Sébastien FERRÉ et Mireille DUCASSÉ pour m'avoir accordé leur confiance et leur soutien pendant mon Master 2, et enfin Arnaud GOTLIEB pour ses encouragements.

Cette thèse n'aurait pu exister dans sa forme actuelle sans les personnes avec qui j'ai eu l'occasion de travailler au cours de ces trois années. En tant que membre du projet VALMEM, j'ai eu la chance de bénéficier d'une étude de cas réaliste et très motivante en la mémoire SPSMALL, grâce aux travaux des autres membres du projet que sont, outre mes deux directeurs de thèse, Abdelrezzak BARA, Pirouz BAZARGAN-SABET, Remy CHEVALLIER, Dominique LE DU et Patricia RENAULT. Cette étude de cas a, par sa complexité, motivé plusieurs techniques développées dans cette thèse, et notamment la réalisation de IMITATOR II. Merci également à Ulrich KÜHNE pour avoir poursuivi le développement d'IMITATOR II, et à Romain SOULAT pour avoir expérimenté de nouvelles techniques permettant ainsi l'analyse de deux modèles de la mémoire SPSMALL (et pour avoir martyrisé `passion` de longues nuits durant). I also thank Jeremy SPROSTON for reading my thesis and suggesting numerous interesting enhancements.

Évidemment, merci au LSV pour la très bonne ambiance qui y règne, et qui permet d'y réaliser une thèse dans d'excellentes conditions. Et puis, comme

il est d'usage de glisser dans cet exercice obligé quelques allusions *private* que personne ne comprendra ou ne lira jamais, probablement pas même les personnes concernées, merci aux occupants de la salle Renodo pour leurs (bruyantes) parties de travail collaboratif du temps où ils tentaient encore de rivaliser avec moi, merci à la RATP de m'avoir permis de réaliser *Ticket II*, qui aura égayé le mur en face de moi à défaut de m'enrichir, merci à mon presque voisin de bureau pour m'avoir emmené à la découverte des paysages ferroviaires d'Asie centrale au beau milieu de l'hiver et de ma thèse et au grand dam de nos encadrants respectifs, merci à Valérie pour avoir contribué à développer nos capacités créatrices et nous avoir fait découvrir tout Paris à pied, merci à mes piments pour avoir consciencieusement poussé, même dans les moments difficiles que représente l'hiver cachannais, merci à ceux (et surtout celles) qui les ont arrosés – et puis, *last but not least* comme on dit, merci à mon vélo. Enfin, merci à mes parents sans qui je ne serais pas là (c'est une tautologie). 最後，非常感謝黃磊長久以來的支持和幫助。

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

When did everything start having
an expiration date?

Chungking Express
(Wong Kar-wai)

The importance of computer systems has dramatically increased in the last decades. Critical systems, involving human lives, need to be perfectly reliable, with a total absence of any inappropriate behavior, such as failures or unexpected sequences of actions. One can test a given system to check the absence of inappropriate behaviors for a given environment, by directly executing the system for this environment; however, although testing a system can show the absence of bad behavior for a given execution, no guarantee is given in the general case, for different scenarios of the environment. Moreover, if there is some nondeterminism in the execution, the correctness of a test case does not give any guarantee, even exactly for the same environment. As a consequence, testing can be used in order to find possible errors in a system, but usually not to show its correctness. This is why formal verification is needed, allowing to prove the correctness of a system with respect to some properties, using mathematical models and proofs.

When considering timed systems, i.e., systems involving time elapsing between actions, correctness also takes timing constraints into account. In a real-time environment, i.e., for systems working in a highly interactive environment, such as hardware devices or telecommunication protocols, it is not sufficient to prove that the system behaves well: one also needs to prove that this good behavior always occurs within a given interval of time. For example, when designing the computer system of a car, one can formally prove that, in case of a collision, the airbag will *eventually* inflate. However, one easily understands that this proof is not enough, and that one should prove that the airbag will inflate within a given amount of time (hopefully short) after the collision occurred. In other words, timed systems have to meet not only constraints on the order of the events, but also quantitative constraints on delays between these events.

Moreover, in a concurrent environment where several timed systems interact with each other, an action occurring too late (or, more generally, at a wrong time) in a subsystem can change the global behavior of the whole system, and have dramatic consequences. For example, the Therac-25 machine, a radiation therapy machine, severely injured and killed several patients in the 1980s because the patients were given by mistake high overdoses of radiation. Beside numerous bugs in the software code and in the development process, one of the causes for this dramatic malfunctioning was a race condition, i.e., an unexpected ordering of actions leading to unpredictable behaviors. The bug in the software allowing this race condition was difficult to detect by simple testing, because it occurred only after a specific sequence of actions within a short amount of time, and the engineers having tested the system were not familiar enough with the machine for them to perform this sequence of actions quickly enough.

1.1 The Good Parameters Problem

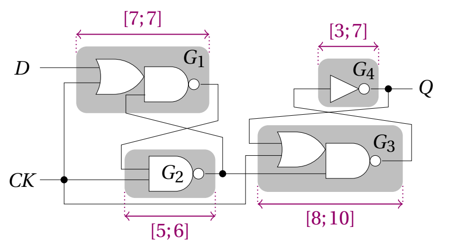
When designing a real-time system, it is not sufficient to prove the correctness for some given delays: one also wants to know for which delays the behavior of the system is correct. Checking the correctness of the system for various values of those delays can be difficult and time consuming. It is therefore interesting to reason parametrically, by considering that these delays are *unknown constants*, or *parameters*. The problem then consists in synthesizing good values for those parameters, i.e., values for which the system is guaranteed to behave well. This can be done by synthesizing a *constraint* on these parameters guaranteeing a correct behavior.

This thesis faces the following good parameters problem, as mentioned in [FJK08] in the framework of hybrid systems: "given a timed system involving delays, find values for those delays seen as parameters within a bounded domain for which the system behaves well".

Finding suitable values for the timing delays is of interest for guaranteeing a good behavior in a concurrent timed system. This allows a notion of robustness: one can guarantee that the values *around* a given value of the delays will not impact the overall behavior of the system, or will still guarantee a (possibly different) good behavior. Moreover, it allows the optimization of some of the timing delays, without changing the overall behavior of the system.

However, this problem is generally difficult for concurrent timed systems. Such systems involve both a functional behavior (i.e., sequences of actions, possibly concurrent) and a timed behavior (i.e., durations between two actions). As a consequence, abstracting either time or functionalities leads in general to too imprecise results. Simple heuristics do generally not apply to this framework: for example, in order to minimize the response (or computation) time of a system, one may want to minimize every (or some) intermediate timing delay. However, it is unfortunately well known (in particular, in hardware verification) that diminishing an intermediate timing delay can actually *increase* the global response or computation time. As a consequence, one should consider *all* possibilities. First, this may not be possible if one considers real-valued timing delays, because this would result in an infinite number of models to verify. Second, even with suitable abstractions allowing to group interval of values together, this may result in the explosion of the state-space if the considered model is too precise.

A Motivating Example. As an example, consider the asynchronous "D flip-flop" circuit described in [CC07] and depicted in Figure 1.1 left. It is composed of 4 elements (G_1, G_2, G_3 and G_4) interconnected in a cyclic way. Elements G_1 and G_3 are made of an "OR" gate and a "NAND" gate. Element G_2 is a single "NAND" gate, and element G_4 is a single "NOT" gate (or inverter). The environment involves two input signals D and CK. The global output signal is Q.



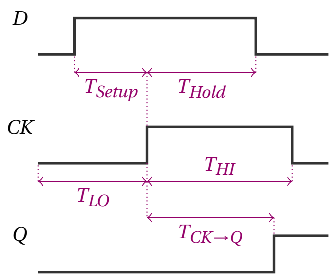


Figure 1.1: Flip-flop circuit (left) and its environment (right)

We consider a bi-bounded inertial model for gates (see [BS95, MP95]), where any change of the input may lead to a change of the output (after some delay). As a consequence, each gate G_i has a timing delay in the parametric interval $[\delta_i^-, \delta_i^+]$, with $\delta_i^- \leq \delta_i^+$. There are 4 other timing parameters (viz., T_{HI}, T_{LO}, T_{Setup}, and T_{Hold}) used to model the environment. The output signal of a gate G_i is denoted by g_i (note that $g_4 = Q$). The rising (resp. falling) edge of signal D is denoted by $D^{\uparrow}$ (resp. $D^{\downarrow}$) and similarly for signals $CK, Q, g_1, \ldots, g_4$.

We consider an environment starting from $D = CK = Q = 0$ and $g_1 = g_2 = g_3 = 1$, with the following ordered sequence of actions for inputs D and CK: $D^{\uparrow}$, $CK^{\uparrow}$, $D^{\downarrow}$, $CK^{\downarrow}$, as depicted in Figure 1.1 right.

We consider that the behavior of this circuit is correct if, for this environment, the rise of signal Q (i.e., action $Q^{\uparrow}$) always occurs before the fall of signal CK (i.e., action $CK^{\downarrow}$).

The question that now arises is: what are the possible values for these 12 timing parameters such that the circuit behaves in a correct way? As said above, this is a difficult problem, in the sense that testing all the possible values for those parameters is simply not possible.

We will develop in this thesis techniques allowing to answer this question.

Timed Automata. We will here mostly focus on the good parameters problem in the framework of timed automata [AD94]. Timed automata are an extension of the class of standard finite-state automata, making use of clocks, which are real-valued variables evolving linearly at the same rate. Those clocks are compared with the delays of the system in constraints that must be verified in order to stay in a state of the automaton, or in order to take a transition. One can also reset some clocks when firing transitions. The model of timed automata has been widely used in order to study and verify hardware devices or communication protocols. However, timed automata can verify the correctness of a system only for one given set of values for the timing parameters.

When facing the problem of synthesis of parameters ensuring the correctness of the system, one needs to consider *parametric* timed automata [AHV93]. Parametric timed automata are an extension of timed automata to the parametric case, allowing in the constraints the use of parameters in place of constants.

Contributions. We propose here a novel approach for solving the good parameters problem in the framework of parametric timed automata. This approach relies on the following *inverse method*: given a reference valuation of the parameters, we synthesize a constraint on the delays viewed as parameters, guaranteeing the same time-abstract behavior as for the reference valuation. Roughly speaking, this time-abstract behavior only relies on actions, and not on the time elapsing between actions. This method has two main advantages. First, it gives a criterion of *robustness* by ensuring the correctness of the system for other values for the parameters around the reference valuation. This is of interest when implementing a system: indeed, the exact model with (for example) integer values for timing delays that has been formally verified will necessarily be implemented using values which will not be exactly the ones that have been verified. Second, it allows the system designer to *optimize* some delays without changing the overall functional behavior of the system.

By iterating this inverse method on various points of a bounded parameter domain, we are then able to separate parameter zones into zones for which the time-abstract behavior of the system is uniform. This gives a *behavioral cartography* of the system. One can then partition those zones into good zones and bad zones, with respect to a given property one wants to verify. The main interest is that this cartography does not depend on the property one wants to verify: only the partition into good and bad zones actually does. As a consequence, when verifying other properties, it is sufficient to check the property for only one point in each zone in order to get the new partition.

Two versions of prototypes have been implemented; in particular, IMITATOR II implements the inverse method, and its behavioral cartography algorithm, in the framework of timed automata. This prototype allowed us to synthesize values for the parameters of several case studies such as abstractions of a memory circuit sold by the chipset manufacturer ST-Microelectronics, or communication protocols.

Both this inverse method and the behavioral cartography algorithm naturally extend to probabilistic timed automata. In this framework, the constraint synthesized guarantees that, for any valuation of the parameters of a constraint synthesized by the inverse method, the minimum (resp. maximum) probabilities of reaching a given state will be the same. Besides the advantages inherited from the non-probabilistic framework, the main practical advantage of this extension is that it allows the *rescaling* of constants. Indeed, when verifying properties on probabilistic timed systems, for example using the PRISM model checker [HKNP06], the verification time and memory may highly depend on the size of the constants of the system. As a consequence, it can sometimes be extremely difficult (or even impossible) to compute probabilities for the "official" constants of a case study, and one has to rescale them in an approximate way, without much guarantee of correctness. By synthesizing a constraint guaranteeing the equality of reachability properties, one may safely rescale the constants of the system, and compute the probabilities for smaller constants (within the constraint).

The underlying principle of the inverse method is not strongly related to the framework of timed automata, and may also be applied to other formalisms. As a consequence, we generalize the synthesis of parameters in two untimed frameworks, namely Directed Weighted Graphs and Markov Decision Processes. In that case, the parameters do not necessarily correspond to timing delays. We extend the inverse method to those two frameworks as follows. First, in the framework of Directed Weighted Graphs, we synthesize a constraint on the costs of the system seen as parameters guaranteeing that the path of optimal cost between any two nodes of a graph will remain the same, for any valuation of the parameters satisfying that constraint. Second, in the framework of Markov Decision Processes, which allow to consider untimed probabilistic systems, we synthesize a constraint on the costs seen as parameters guaranteeing that the optimal policy of the system remains the same, for any valuation of the parameters satisfying that constraint. Prototypes have been implemented.

1.2 Context

This thesis focuses on the verification and synthesis of parameters in the framework of real-time systems. This problem has applications in various domains such as software analysis, hardware analysis, or networking protocols, and can be used for industrial applications in the medical industry, aeronautics, biology, etc. We develop hereafter some of the classical problems and techniques related to this problem.

1.2.1 Classical Problems

Computation of Response Time. When verifying a real-time system, one of the major problems is the computation of the *response time* of the system. This classical problem consists, given an environment, i.e., a scenario for the inputs of the system, in computing the time between the change of the inputs and the change of a given output. This computation of response time is critical in particular when considering the verification of hardware, and more specifically of asynchronous circuits. An approach to compute the response time of a system is the Time Separation of Events (see, e.g., [CDY01] for an extensive survey). As said in [CDY01]: “The behavior of asynchronous and concurrent systems is naturally described in terms of events and their interactions. A fundamental problem in analyzing such systems is to determine bounds on the time separation of events. Stated informally, we seek answers to questions such as: How late can event i occur after event j? for arbitrary events i and j. The problem of computing time separation bounds is compounded in practice by statistical variations in manufacturing and operating conditions that introduce uncertainties in component delays. Consequently, finding bounds on time separation of events in the presence of uncertain component delays is an important practical problem.” The system can be represented under the form of an (oriented) timing constraint graph [CSD97], where nodes represent events, and the directed edges represent dependencies between them. Various techniques have been proposed to solve this problem, either exact or using approximations (see, e.g., [CDY01, EF08] for a survey).

Worst-Case Execution Time. A strongly related problem is the *worst-case execution time* (WCET) of some computation of a hardware device. It corresponds to the maximum length of time it takes to execute a given task on this device, and thus gives an upper bound on the execution time. Although it is well-known that giving an upper bound on the execution of a program (in the general sense) is impossible, one can compute such an upper bound on real-time systems, which can be seen as a restricted form of programming [WEE+08]. In most cases, the state space is too large to exhaustively explore all possible executions and thus determine the exact worst-case execution time. As a consequence, one needs to define methods using abstractions or overapproximations, with the smallest possible loss of precision. Various techniques can be used; among them, one can first cite static program analysis [CC77, NNH99], which creates an abstraction of the device while avoiding to actually execute it. Second, (exact) simulation is a classical technique to estimate execution time for hardware verification using an exact model. A survey on techniques for WCET analysis can be found in [WEE+08], providing in particular a thorough overview of dedicated tools.

Sensitivity Analysis. Beside the verification of concurrent timed systems, we are also interested in the synthesis of parameters. In particular, our inverse method aims at synthesizing parameter valuations around a reference valuation, and corresponding to the same behavior. This problem is related to the *sensitivity analysis*, i.e., the study of the variation of some inputs on the global behavior of a system. This domain has very large applications in areas such as finance, mathematics, chemistry, environment, etc. In particular, it allows to give a condition of *robustness* to the system, by finding sets of parameters or input behaviors for which the global behavior of the system will remain (relatively) unchanged.

Although, to our knowledge, no attempt of performing a sensitivity analysis on timed automata has been performed, the notion of robust timed automata should be mentioned: this approach guarantees the good behavior of timed automata even for small variations (or *drifts*) of the clocks. Robust timed automata will be discussed in Section 2.4.2.

We will also extend our work on timed systems to the *untimed* world, and more precisely to directed weighted graphs (see Section 7.1). In that case, we get much closer to the sensitivity analysis community. In particular, the authors of [ROC05] address a very similar problem to the problem we face in Section 7.1. We will be interested in synthesizing constraints on the weights of directed weighted graphs, such that the shortest paths of the graph all remain the same as for a reference valuation of the weights. The main difference with the work of [ROC05] is that we reason *parametrically*. Indeed, whereas the authors of [ROC05] determine the maximum and minimum weights that each edge can have so that a given path remains optimal, we are able to infer relations under the form of a *constraint* between the weights of the graph. More will be discussed in Section 7.3.

1.2.2 Formal Techniques of Verification

Testing the correctness of a system by simulation is often performed when verifying the correctness of a system in practice. However, simulation and testing techniques only consider specific environments, and cannot give guarantees on *all* the possible executions. As a consequence, those techniques are generally inadequate for the verification of critical systems. One needs formal techniques, involving formal semantics, modeling languages, specification languages, and verification algorithms.

Model Checking. The model checking approach considers a system and a property to check on the system, and builds on the one hand a mathematical model of the system, and on the other hand a mathematical model for the property [EC80, CE82]. Techniques are then applied, allowing to *automatically* check whether the mathematical model of the system satisfies the mathematical model of the property. If yes, we say that the system meets the specification. Mathematical models for systems can be oriented graphs, i.e., automata and their extension (including timed automata [AD94]), as well as Petri nets [Pet62, Mer74] or process algebra. Properties can also be modeled by various structures, such as extensions of automata (including timed automata) or logics, including temporal logics (e.g., [Pnu77, CE82, ACD93]).

Temporal Logics. Temporal logics are extensions of propositional logic, allowing to specify the behavior of a reactive system over time using temporal modalities. Those logics allow to specify *order* on events, but generally not to introduce timing constraints specifying the time value at which an event must occur. Temporal logics allow in particular the specification of properties such as reachability (possibility that a certain event occurs), safety (impossibility that a certain set of events occurs), liveness (ultimate occurrence of a certain event) or fairness (occurrence infinitely often of a certain event). Temporal logics can be either *linear* or *branching*.

Linear Temporal Logic (LTL) is a logic introduced by Pnueli [Pnu77]. It is linear in the sense that it considers that time steps have only one (discrete) successor. Therefore, it is possible to express ordering of events on single paths, and not on tree structures: LTL expresses *path-based properties*. Various variants and extensions of LTL have been studied (see, e.g., [GPSS80, LPZ85, Lam94]). The logic LTL is widely used in the framework of model checking, and various algorithms have been proposed. An efficient model checker allowing to check properties expressed using LTL is Spin [Hol03].

We now consider *branching* temporal logics. Whereas linear temporal logics focus on infinite sequence on states, branching logics focus on infinite trees of states. As a consequence, properties specified using branching temporal logics can express the notion of *choice*, in the sense that one can express the different possible futures of a given state. In particular, CTL [CE82] is a widely used branching temporal logic. As for LTL, CTL is widely used in the framework of model checking, and various algorithms have been proposed. Model checkers include SMV [McM93] (which stands for Symbolic Model Verifier) and NuSMV [CCG$^+$02]. It can be shown that the expressiveness of LTL and CTL is incomparable, i.e., some properties can be expressed using LTL only, and some others using CTL only (see, e.g., [BK08] for a survey).

We can consider that temporal logics such as LTL and CTL are *time-abstract* logics, in the sense that they only focus on the order of events, and not on the precise time at which they occur. One can then extend the temporal logics with *time*, thus obtaining *timed* temporal logics. Those logics do not only check that constraints on the order of the events are satisfied, but also that quantitative constraints on delays between these events are satisfied. A famous example of timed temporal logic is the timed CTL (TCTL) logic [ACD93], which allows to express properties specifying both a branching behavior and interval of time within which events may occur. The TCTL logic is widely used in the framework of timed model checking, in particular to verify properties on models expressed using timed automata. Powerful model checkers for (fragments of) TCTL include UPPAAL [LPY97] and KRONOS [Yov97].

Formalisms for Modeling Timed Systems. Structures for the modeling and the verification of real-time systems include Time Petri Nets [Mer74], Timed Automata, Time Separation of Events using timing constraint graphs [CSD97], and various extensions. We provide a survey on various formalisms for the modeling of timed systems in Section 2.4.

1.3 Organization of the Thesis

This thesis is structured as follows.

In Chapter 2, we recall the major formalisms used in this thesis. We first recall the notion of clocks, parameters and constraints. We then recall timed automata, and their extension to parametric timed automata. We also formally state the good parameters problem we intend to solve in this thesis in the framework of parametric timed automata. We finally justify our choice of the formalism of timed automata, and present its specificities with respect to other timed formalisms (in particular Time Petri Nets).

In Chapter 3, we introduce the inverse method, which allows to generalize the behavior of a timed automaton by synthesizing a constraint on the parameters guaranteeing the same time-abstract behavior.

In Chapter 4, we present our implementation, the tool IMITATOR II, and we apply our method to various case studies of asynchronous hardware circuits and communication protocols.

In Chapter 5, we show how an iteration of the inverse method can solve the good parameters problem for parametric timed automata, by computing a behavioral cartography of the system. We also apply this algorithm to various case studies using IMITATOR II.

In Chapter 6, we extend our inverse method, and the behavioral cartography, to the framework of probabilistic timed automata. In that case, we synthesize a constraint guaranteeing the same values for the minimum and maximum probabilities of reachability properties.

In Chapter 7, we introduce variants of the inverse method for two different frameworks: Directed Weighted Graphs, and Markov Decision Processes.

We finally conclude and present directions of future research in Chapter 8.

Related work is mentioned at the end of each chapter.

VALMEM Project. This thesis has been done in the framework of the ANR VALMEM project "Functional and Timed Validation of Embedded Memories Using Formal Methods", grant ANR-06-ARFU-005, involving the LIP 6 laboratory (Université Pierre et Marie Curie), the LSV laboratory (École Normale Supérieure de Cachan), and the chipset manufacturer ST-Microelectronics. Many techniques developed in this thesis have been actually designed in order to be able to synthesize constraints to guarantee the good behavior of various models and abstractions of the SPSMALL memory, which is a memory circuit sold by ST-Microelectronics. Due to its internal complexity, this memory has been a very interesting motivation throughout my thesis.

Joint Work. Most of this work is a joint work with Laurent Fribourg and Emmanuelle Encrenaz. Thomas Chatain collaborated on the inverse method for parametric timed automata (Chapter 3). Jeremy Sproston collaborated on the extension of the inverse method to the framework of probabilistic timed automata (Chapter 6). The fixpoint of the inverse method benefited from discussion with Laurent Doyen. The counter-example showing the non-CTL equivalence of the inverse method was proposed by Jeremy Sproston.

The analysis of the SPSMALL memory (Section 4.7) has been done in the framework of the ANR VALMEM project involving (besides Emmanuelle Encrenaz, Laurent Fribourg and me) Remy Chevallier (from ST-Microelectronics), Abdelrezzak Bara, Pirouz Bazargan-Sabet, Dominique Le Du and Patricia Renault (from LIP 6). The analysis of the SIMOP Networked Automation System (Section 4.8) has been done in the framework of the SIMOP project in the framework of Institut Farman (Fédération de Recherche CNRS, FR3311), with the contribution of Olivier De Smet, Bruno Denis and Silvain Ruel (LURPA, École Normale Supérieure de Cachan).

When coming to the design and the implementation of the tool IMITATOR II (Chapter 4), Ulrich Kühne implemented several modifications allowing to decrease the computation time. Bertrand Jeannet has been of great help when linking the tool with the APRON library. Daphné Dussaud implemented the graphical output of the cartography. Romain Soulat has been a great contributor of IMITATOR II by applying the tool to various case studies; in particular, he applied the cartography algorithm implemented in IMITATOR II to the SPSMALL memory, using a very helpful optimization of his own.

Chapter 2

Preliminary Definitions

You're entering a world of pain.

The Big Lebowski
(Joel and Ethan Coen)

In this chapter, we present the formalism used throughout this thesis. We focus on a way to model timed systems by means of timed automata [AD94]. Timed automata are an extension of standard finite-state automata allowing the use of clocks, i.e., real-valued variables increasing linearly at the same rate. Such clocks can be compared with constants in constraints that allow (or not) to stay in a location ("invariants") or to take a transition ("guards"). At each transition, it is possible to reset some of the clocks of the system. This formalism allows the parallel composition of several timed automata, which behave like a single one, and thus provides the designer with a powerful and intuitive way to represent timed systems. Above all, the main theoretical advantage of timed automata relies in its decidability results. In particular, it has been shown that the reachability of a state is decidable. Moreover, various *timed* temporal logics (e.g., [ACD93]) have been designed, and various decidability results have been shown (e.g., [ACD93, HRSV02, WY03]). Finally, it is important to note that the model of Timed Automata is very sensitive to the size of the automata and the number of automata in parallel, thus often leading to the state-space explosion problem. However, powerful tools, such as the UPPAAL [LPY97] model checker, have been designed allowing to model and verify very efficiently timed systems modeled by Timed Automata.

We are interested in this thesis in synthesizing values for timing parameters of a system, guaranteeing a good behavior. As a consequence, we will use a parametric extension of timed automata. Those *parametric timed automata* [AHV93] allow in guards and invariants the use of parameters (in the sense of unknown constants) in place of rational constants. Unfortunately, for most interesting problems, parametric timed automata lose the decidability results proved for timed automata. In particular, the reachability of a state is not decidable (although semi-algorithms do exist, i.e., if the algorithm terminates, then the result is correct). Moreover, parametric timed automata are even more sensitive to the state space explosion problem, because of the addition of the parameters. In practice, this comes also from the fact that the data structures used to represent parametric timed automata are far less efficient than the ones used for timed automata (typically Difference Bound Matrices, proposed in [BM83] for the analysis of Time Petri Nets, and introduced in [Dil90] for Timed Automata). Structures allowing to handle parametric timed models include Parametric Difference Bound Matrices (an extension of Difference Bound Matrices, proposed in [HRSV02]), SAT-solvers, SMT-solvers and polyhedra. Avoiding the explosion of the state space, and finding cases for which analyses are decidable for parametric timed automata, are actually some of the motivations for this thesis.

We will also state formally in this chapter the good parameters problem in the framework of parametric timed automata. This problem can be considered as the central problem of this thesis.

Plan of the chapter. We first recall in Section 2.1 the notion of linear constraints on the clocks and the parameters. We then describe in Section 2.2 the formalism of Timed Automata, and its extension to Parametric Timed Automata. We introduce in Section 2.3 the good parameters problem that we will intend to solve in this thesis. We finally give in Section 2.4 a survey of formalisms used to model timed systems, and justify our choice for dense time representation and parametric timed automata.

2.1 Constraints

2.1.1 Clocks

Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers.

Throughout this thesis, we assume a fixed set $X = \{x_1, \ldots, x_H\}$ of *clocks*. A *clock* is a variable x_i with value in $\mathbb{R}_{\geq 0}$. All clocks evolve linearly at the same rate.

We define a *clock valuation* as a function $w : X \to \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each clock variable. We will often identify a valuation w with the point $(w(x_1), \ldots, w(x_H))$.

Given a constant $d \in \mathbb{R}_{\geq 0}$, we use $X + d$ to denote the set $\{x_1 + d, \dots, x_H + d\}$. Similarly, we write $w + d$ to denote the valuation such that $(w+d)(x) = w(x) + d$ for all $x \in X$.

2.1.2 Parameters

Throughout this thesis, we assume a fixed set $P = \{p_1, \dots, p_M\}$ of *parameters*, i.e., unknown constants.

A *parameter valuation* π is a function $\pi : P \to \mathbb{R}_{\geq 0}$ assigning a nonnegative real value to each parameter. There is a one-to-one correspondence between valuations and points in $(\mathbb{R}_{\geq 0})^M$. We will often identify a valuation π with the point $(\pi(p_1), \dots, \pi(p_M))$.

2.1.3 Constraints

Definition 2.1 (Linear inequality)**.** Let V be a set of variables of the form $V = \{v_1, \dots, v_N\}$. A *linear inequality on the variables of* V is an inequality $e \prec e'$, where $\prec \in \{<, \leq\}$, and e, e' are two linear terms of the form

$$\sum_{1 \leq i \leq N} \alpha_i v_i + d$$

where $v_i \in V$, $\alpha_i \in \mathbb{Q}_{\geq 0}$, for $1 \leq i \leq N$, and $d \in \mathbb{Q}_{\geq 0}$. ■

Note that we define the coefficients of the linear inequalities as *positive rationals*. It would of course be equivalent to define them as *positive integers*, as it is sometimes the case in the literature.

We assume in the following that all inequalities are linear, and we will simply refer to linear inequalities as *inequalities*.

Definition 2.2 (Negation of an inequality)**.** Let V be a set of variables of the form $V = \{v_1, \dots, v_N\}$. Given an inequality J on the variables of V of the form $e < e'$ (resp. $e \leq e'$), the *negation* of J, denoted by $\neg J$, is the linear inequality $e' \leq e$ (resp. $e' < e$). ■

Definition 2.3 (Convex linear constraint)**.** Let V be a set of variables of the form $V = \{v_1, \dots, v_N\}$. A *convex linear constraint on the variables of* V is a conjunction of inequalities on the variables of V. ■

We assume in the following that all constraints are both convex and linear, and we will simply refer to convex linear constraints as *constraints*.

Definition 2.4 (Constraint on the clocks)**.** An *inequality on the clocks* is an inequality on the set of variables X. A *constraint on the clocks* is a constraint on the set of variables X. ■

Definition 2.5 (Constraint on the parameters)**.** An *inequality on the parameters* is an inequality on the set of variables P. A *constraint on the parameters* is a constraint on the set of variables P. ■

Definition 2.6 (Constraint on clocks and parameters)**.** An *inequality on the clocks and the parameters* is an inequality on the set of variables $X \cup P$. A *constraint on the clocks and the parameters* is a constraint on the set of variables $X \cup P$. ■

Throughout this thesis, we will denote by $\mathcal{K}_X$ the set of all constraints on the clocks, by $\mathcal{K}_P$ the set of all constraints on the parameters, and by $\mathcal{K}_{X \cup P}$ the set of all constraints on the clocks and the parameters.

In the sequel, the letter J will denote an inequality on the parameters, the letter D will denote a constraint on the clocks, the letter K will denote a constraint on the parameters, and the letter C will denote a constraint on the clocks and the parameters.

Semantics of Constraints. Given a constraint D on the clocks and a clock valuation w, $D[w]$ denotes the expression obtained by replacing each clock x in D with $w(x)$. A clock valuation w *satisfies* constraint D (denoted by $w \models D$) if $D[w]$ evaluates to true.

Given a parameter valuation π and a constraint C on the clocks and the parameters, $C[\pi]$ denotes the constraint on the clocks obtained by replacing each parameter p in C with $\pi(p)$. Likewise, given a clock valuation w, $C[\pi][w]$ denotes the expression obtained by replacing each clock x in $C[\pi]$ with $w(x)$. We say that a parameter valuation π *satisfies* a constraint C, denoted by $\pi \models C$, if the set of clock valuations that satisfy $C[\pi]$ is nonempty. We use the notation $<w,\pi> \models C$ to indicate that $C[\pi][w]$ evaluates to true.

Given two constraints C_1 and C_2 on the clocks and the parameters, we say that C_1 is *included in* C_2, denoted by $C_1 \subseteq C_2$, if $\forall\, w,\pi : <w,\pi> \models C_1 \Rightarrow <w,\pi> \models C_2$. We have that $C_1 = C_2$ if and only if $C_1 \subseteq C_2$ and $C_2 \subseteq C_1$.

Similarly to the semantics of constraints on the clocks and the parameters, we say that a parameter valuation π *satisfies* a constraint K on the parameters, denoted by $\pi \models K$, if the expression obtained by replacing each parameter p in K with $\pi(p)$ evaluates to true. Given two constraints K_1 and K_2 on the parameters, we say that K_1 is *included in* K_2, denoted by $K_1 \subseteq K_2$, if $\forall \pi : \pi \models K_1 \Rightarrow \pi \models K_2$. We have that $K_1 = K_2$ if and only if $K_1 \subseteq K_2$ and $K_2 \subseteq K_1$. We will consider `true` as a constraint on the parameters, corresponding to the set of all possible values for P.

Given a constraint C on the clocks and the parameters, we denote by $\exists X : C$ the constraint on the parameters obtained from C after elimination of the clock variables, i.e., $\{\pi \mid \exists w : <w,\pi> \models C\}$.

2.2 Timed Automata

2.2.1 Labeled Transition Systems

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

Definition 2.7 (LTS)**.** A *labeled transition system (LTS)* over a set of symbols Σ is a triple $\mathscr{L} = (S, S_0, \Rightarrow)$, with S a set of *states*, $S_0 \subset S$ 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 $\mathscr{L}$ is a finite 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. ■

2.2.2 Timed Automata

We introduce here timed automata as defined in [AD94]. Timed automata are an extension of standard finite-state automata allowing the use of clocks, i.e., real-valued variables increasing linearly at the same rate.

Syntax

Definition 2.8 (Timed Automaton)**.** A *Timed Automaton (TA)* $\mathscr{A}$ is a 6-tuple of the form $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$, where

- Σ is a finite set of actions,
- Q is a finite set of locations,
- $q_0 \in Q$ is the initial location,
- X is a set of clocks,
- I is the invariant, assigning to every $q \in Q$ a constraint $I(q)$ on the clocks, and
- $\rightarrow$ is a step relation consisting of elements of the form (q, g, a, ρ, q'), also denoted by $q \overset{g,a,\rho}{\rightarrow} q'$, where $q, q' \in Q$, $a \in \Sigma$, $\rho \subseteq X$ is a set of clock variables to be reset by the step, and g (the step guard) is a constraint on the clocks.

■

Note that we use a more permissive definition of the constraints used in guards and invariants than in the original definition of TAs (see [AD94]). Indeed, we allow the use of conjunctions of any linear inequalities on the clocks, whereas the original definition usually considers conjunctions of comparisons of a single clock with a constant. This more permissive definition has usually an impact on the decidability (the addition of clock values within a constraint leads to undecidability [AD94]), but this has no impact in this thesis, mainly because of the use of *parametric* timed automata, where the parameters bring themselves undecidability in the general case.

Timed Automata are often extended in practice with *discrete variables*, which can be used in guards and transitions, updated within the transitions, and sometimes even used as a factor for clocks. However, in most cases, there represent only syntactic sugar for the discrete space (i.e., locations). As a consequence, we will not use them in any theoretical part of this thesis. Note nevertheless that our implementation (see Chapter 4) allows the use of such discrete variables.

The graphical representation of a TA $\mathscr{A}$ is an oriented graph where vertices correspond to locations, and edges correspond to actions of $\mathscr{A}$. We follow the following conventions for the graphical representation of timed automata: locations are represented by nodes, above of which the invariant of the location is written; transitions are represented by arcs from one location to another location, labeled by the associated guard, the action name and the set of clocks to be reset (guards and invariants equal to `true` will be omitted). The initial location is usually represented with a double circle.

Example 2.9. We give in Figure 2.1 an example of Timed Automaton containing 4 locations (viz., q_0, q_1, q_2 and q_3), 3 actions (viz., a, b and c) and 2 clocks (viz., x_1 and x_2). The initial location is q_0.

In this TA, q_0 has invariant $x_1 \leq 5$, q_1 has invariant `true`, and both q_2 and q_3 have invariant $x_2 \leq 5$. The transition from q_0 to q_1 has guard $x_1 \geq 4$ through action a; no clock is reset. The transition from q_0 to q_2 has guard $x_1 \geq 2 \wedge x_2 \geq 3$ through action b, and resets clock x_2. The transitions between q_2 and q_3 can be explained similarly.

□

Parallel Composition of TAs

We now introduce the notion of network of timed automata, and show in the following definition how N TAs can be composed into a single TA.

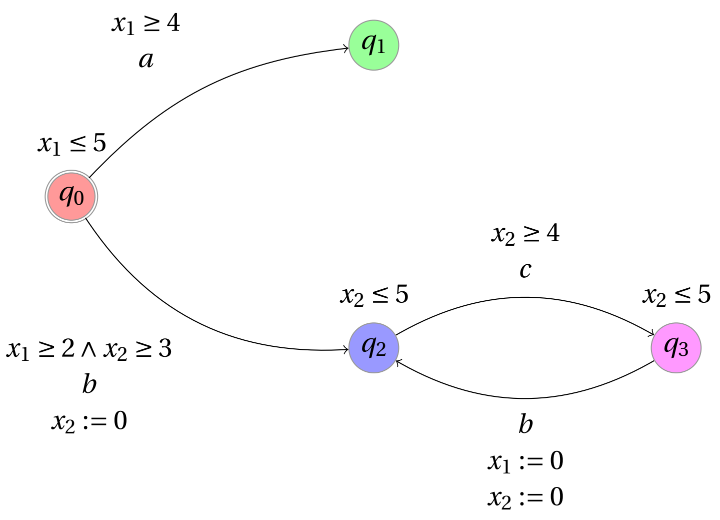


Figure 2.1: An example of Timed Automaton

Definition 2.10 (Network of TAs)**.** Let $N \in \mathbb{N}$. For all $1 \leq i \leq N$, let $\mathscr{A}_i = (\Sigma_i, Q_i, (q_0)_i, X_i, I_i, \rightarrow_i)$ be a TA. The sets Q_i and X_i are mutually disjoint. A *network of timed automata (NTA)* is $\mathscr{A} = \mathscr{A}_1 \| \ldots \| \mathscr{A}_N$, where $\|$ is the operator for parallel composition defined in the following way. This NTA corresponds to the TA $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$ where

- $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,
- $Q = \Pi_{i=1}^{N} Q_i$,
- $q_0 = \langle (q_0)_1, \ldots, (q_0)_N \rangle$,
- $X = \biguplus_{i=1}^{N} X_i$,
- $I(\langle q_1, \ldots, q_N \rangle) = \bigwedge_{i=1}^{N} I_i(q_i)$ for all $\langle q_1, \ldots, q_N \rangle \in Q$,

and $\rightarrow$ is defined as follows. For all $a \in \Sigma$, let T_a be the subset of indices $i \in 1, \ldots, N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $\langle q_1, \ldots, q_N \rangle \in Q$, for all $\langle q'_1, \ldots, q'_N \rangle \in Q$, $(\langle q_1, \ldots, q_N \rangle, g, a, \rho, \langle q'_1, \ldots, q'_N \rangle) \in \rightarrow$ if:

- for all $i \in T_a$, there exist g_i, ρ_i such that $(q_i, g_i, a, \rho_i, q'_i) \in \rightarrow_i$, $g = \bigwedge_{i \in T_a} g_i$, $\rho = \bigcup_{i \in T_a} \rho_i$, and,
- for all $i \notin T_a$, $q'_i = q_i$.

■

Note that the requirement that the set of clocks of each of the TAs in parallel be mutually disjoint is not a strong requirement from a theoretical point of view in this thesis. However, it is almost always met in practice.

Example 2.11. We give in Figure 2.2 an example of Network of 2 TAs. The left one (say, $\mathscr{A}'$) contains 3 locations (viz., q'_0, q'_1, and q'_2), 2 actions (viz., a and b) and 1 clock x_1. The right one (say, $\mathscr{A}''$) contains 3 locations (viz., q''_0, q''_2, and q''_3), 2 actions (viz., b and c) and 1 clock x_2.

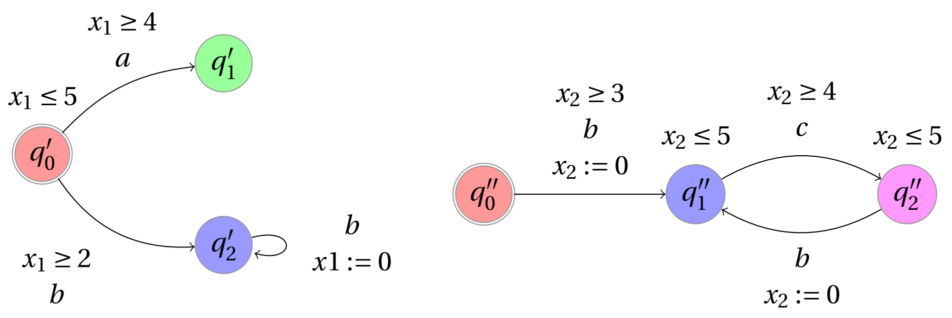


Figure 2.2: Example of Network of Timed Automata

Then, the composition of those 2 TAs in parallel (viz., $\mathscr{A}' \| \mathscr{A}''$) corresponds to the TA from Example 2.9, where $q_0 = (q'_0, q''_0)$, $q_1 = (q'_1, q''_0)$, $q_2 = (q'_2, q''_1)$ and $q_3 = (q'_2, q''_2)$.

□

Semantics

The semantics of TAs is given under the form of a LTS, where states are pairs made by a location and a valuation for each clock.

Definition 2.12 (Semantics of TAs)**.** Let $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$ be a Timed Automaton. The *concrete semantics of* $\mathscr{A}$ is the LTS $(S, S_0, \Rightarrow)$ over Σ where

$$\begin{aligned} S &= \{(q, w) \in Q \times (X \rightarrow \mathbb{R}_{\geq 0}) \mid w \models I(q)\}, \\ S_0 &= \{(q_0, w) \mid w \models I(q_0) \wedge w = (w_0, \ldots, w_0) \text{ for some } w_0\} \end{aligned}$$

and the transition predicate $\Rightarrow$ is specified by the following three rules. For all $(q, w), (q', w') \in S, d \geq 0$ and $a \in \Sigma$,

– $(q, w) \xrightarrow{a} (q', w')$ if $\exists g, \rho : q \overset{g,a,\rho}{\rightarrow} q'$ and $w \models g$ and $w' = \rho(w)$;
– $(q, w) \xrightarrow{d} (q', w')$ if $q' = q$ and $w' = w + d$;
– $(q, w) \overset{a}{\Rightarrow} (q', w')$ if $\exists d, w'' : (q, w) \xrightarrow{a} (q', w'') \xrightarrow{d} (q', w')$. ■

We consider with the definition of S_0 that all clocks are initially set to 0, or have evolved linearly in the bounds given by $I(q_0)$. A state (resp. run) in the concrete semantics will be referred to as a *concrete state* (resp. *concrete run*).

A concrete run is represented by a directed graph where states are depicted within nodes containing the name of the location and the value of each of the clocks, and transitions are depicted using edges labeled with the name of the action.

Example 2.13. Consider again the timed automaton $\mathscr{A}$ of Example 2.1. Then Figure 2.3 depicts an example of concrete run for $\mathscr{A}$.

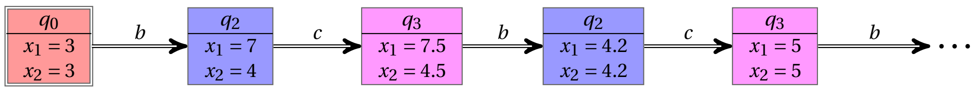


Figure 2.3: Example of concrete run for a TA

This run is obtained as follows: one starts from the initial location q_0 where both clocks have evolved during 3 time units. Then, we take action b, reset x_2 and spend 4 time units in q_2. Then, we take action c, and spend 0.5 time unit in q_3. Then, we take action b, reset both clocks and spend 4.2 time units in q_2. Then, we take action c, and spend 0.8 time units in q_3, and so on.

$\square$

The power of timed automata relies in the fact that one can construct a finite partition of the infinite space of clock valuations. In particular, this construction is suitable to perform reachability analysis.

The main theoretical advantage of Timed Automata relies in its decidability results. In particular, it has been shown that the reachability of a state is decidable. Moreover, various *timed* temporal logics (e.g., [ACD93]) have been designed, and various decidability results have been shown (e.g., [ACD93, HRSV02, WY03]).

Traces

We now introduce the notion of *trace*. This notion allows to analyze a system by abstracting part of its behavior. In the literature, one usually considers either a state-based approach or an action-based approach (see, e.g., [BK08]). We consider here a combined state- and action-based approach: a trace is an alternating sequence of locations and actions.

Note that, for a deterministic TA, i.e., a TA such that there is at most one transition leaving a given location with a given action, there is an equivalence between the state-based, the action-based and the combined approaches (because of the unicity of the initial location). This can be the case for hardware verification when one models circuits at the gate level. Indeed, when one models each gate of the circuit with a different TA, where each location corresponds to a different value of the input and output signals of the gate, and each transition corresponds to a rise or a fall of a signal of the global system, then the composition of the TAs modeling each gate is deterministic. In that case, if one considers a sequence of locations, it is possible to retrieve the corresponding sequence of actions (from a given initial location), and conversely.

We define more formally the notion of trace in the following definition.

Definition 2.14 (Trace associated with a concrete run)**.** Given a TA $\mathscr{A}$ and a concrete run r of $\mathscr{A}$ of the form $(q_0, w_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (q_m, w_m)$, the *trace associated with r* is the alternating sequence of locations and actions $q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m$. We say that location q_i, for $1 \leq i \leq m$, *belongs* to the trace. ■

Note that a trace is built from a run by removing the valuation of the clock, and therefore can be seen as a *time-abstract run*. We depict traces under a graphical form using boxed nodes labeled with locations and double arrows labeled with actions.

Example 2.15. The trace associated with the concrete run of Example 2.13 is depicted in Figure 2.4.

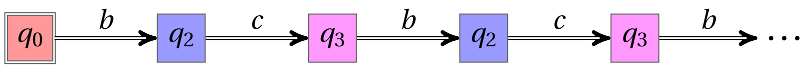


Figure 2.4: Example of trace associated with a concrete run for a TA

□

We define below the notion of acyclic trace as a trace which never passes twice by the same location, i.e., a trace whose locations are all different.

Definition 2.16 (Acyclic trace)**.** Given a trace $T = q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m$, T is said to be an *acylic trace* if:

$$\forall q_i, q_j, i < j < m, q_i \neq q_j$$

■

Given two traces, we define the following notion of prefix of a trace.

Definition 2.17 (Prefix of a trace)**.** Given a trace $T = q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m$, the *prefix of length n of T* is the trace denoted by $|T|_n$ and defined as follows:

$$|T|_n = \begin{cases} q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{n-1}}{\Rightarrow} q_n & \text{if } n < m \\ q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m & \text{else} \end{cases}$$

Similarly, we say that a trace T_1 is a *prefix* of a trace T_2 if there exists $n \geq 0$ such that $|T_2|_n = T_1$. ■

We now define the following notion of trace set.

Definition 2.18 (Trace set)**.** Given a TA $\mathscr{A}$, the *trace set of* $\mathscr{A}$ refers to the set of traces associated with the runs of $\mathscr{A}$. ■

Often, when depicting trace sets, we will not depict each trace separately, but depict the trace set under the form of a tree or a graph. Note, however, that this graph structure is only used for the sake of simplicity of representation of the possible traces, and does not contain any information on the possible *branching* behavior of the system.

Example 2.19. The trace set associated with the TA of Example 2.9 is depicted in Figure 2.5.

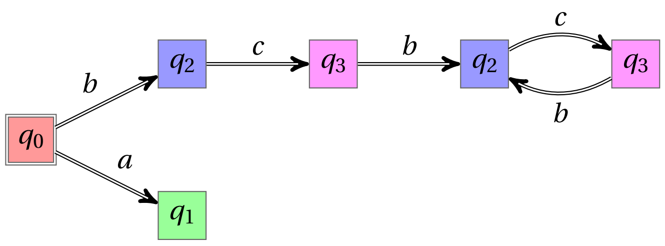


Figure 2.5: Example of trace set of a TA

This trace set contains an infinite number of finite traces. Note also that it is obviously not acyclic, because there are (actually infinitely many) traces passing several times through locations q_2 and q_3.

□

We extend the notion of acyclicity of a trace to trace sets, and say that a trace set is *acyclic* if all its traces are acyclic. We also say that a location q *belongs* to the trace set of $\mathscr{A}$ if it belongs to a trace of the trace set of $\mathscr{A}$.

In the following, we are interested in verifying properties on the trace set of $\mathscr{A}$. For example, given a predefined set of "bad locations", a reachability property is satisfied by a trace if this trace never contains a bad location; such a trace is "good" w.r.t. this reachability property. A trace can also be said to be "good" if a given action always occurs before another one within the trace (see example in Section 3.1.1).

Definition 2.20 (Good trace set)**.** Given a TA $\mathscr{A}$, and a property on traces, we say that a trace of $\mathscr{A}$ is *good* if it satisfies the property, and *bad* otherwise. Likewise, we say that the trace set of $\mathscr{A}$ is *good* if all its traces are good, and bad otherwise.

■

Actually, the good behaviors that can be captured with trace sets are relevant to *linear-time properties* [BK08], which can express properties more general than reachability properties. A more precise characterization of those properties will be done in Section 3.3.3.

2.2.3 Parametric Timed Automata

Parametric Timed Automata are an extension of the class of Timed Automata to the parametric case. Parametric timed automata allow within guards and invariants the use of parameters in place of constants [AHV93]. This 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. The model of Parametric Timed Automata is also interesting to synthesize parameters for which a given property is satisfied.

Syntax

Definition 2.21 (Parametric Timed Automaton)**.** A *parametric timed automaton (PTA)* $\mathscr{A}$ is a 8-tuple of the form $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$, where

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

■

The constraint K on the parameters corresponds to the *initial* constraint on the parameters, i.e., a constraint that will be true in all the states of the PTA (see semantics in Definition 2.31). In the following, given a PTA $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$, we will often denote this PTA by $\mathscr{A}(K)$ when clear from the context, in order to emphasize that only K will change in $\mathscr{A}$.

We make use for PTAs of the same graphical representation as for TAs, i.e., an oriented graph where vertices correspond to the locations, and edges correspond to the actions. The graphical representation of a PTA will be referred to as its *associated graph.*

Example 2.22. We give in Figure 2.6 an example of PTA containing 4 locations (viz., q_0, q_1, q_2 and q_3), 3 actions (viz., a, b and c), 2 clocks (viz., x_1 and x_2) and 2 parameters (viz., p_1 and p_2). The initial location is q_0.

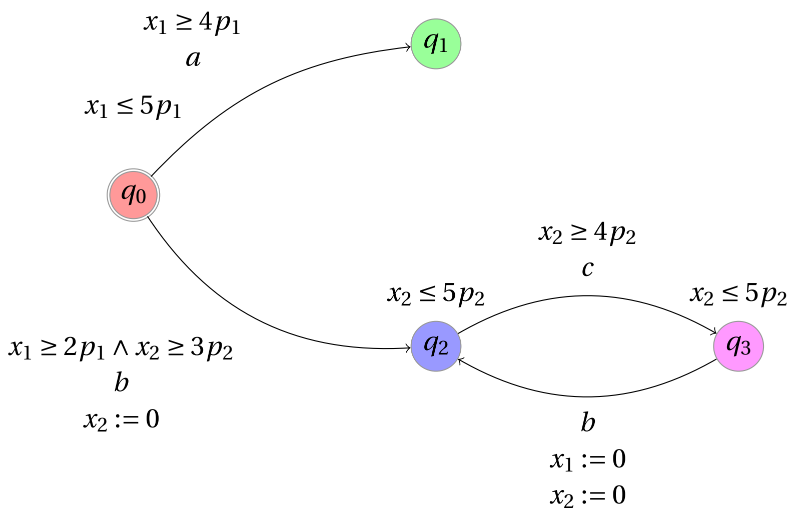


Figure 2.6: An example of Parametric Timed Automaton

In this PTA, q_0 has invariant $x_1 \leq 5p_1$, q_1 has invariant `true`, and both q_2 and q_3 have invariant $x_2 \leq 5p_2$. The transition from q_0 to q_1 has guard $x_1 \geq 4p_1$ through action a; no clock is reset. The transition from q_0 to q_2 has guard $x_1 \geq 2p_1 \wedge x_2 \geq 3p_2$ through action b, and resets clock x_2. The transitions between q_2 and q_3 can be explained similarly. □

Instantiation of a PTA. Given a PTA $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$, for every parameter valuation $\pi = (\pi_1, \ldots, \pi_M)$, $\mathscr{A}[\pi]$ denotes the PTA $\mathscr{A}(K)$, where K is $\bigwedge_{i=1}^{M} p_i = \pi_i$. This corresponds to the PTA obtained from $\mathscr{A}$ by substituting every occurrence of a parameter p_i by constant π_i in the guards and invariants. We say that p_i is *instantiated* with π_i. Note that, as all parameters are instantiated, $\mathscr{A}[\pi]$ is a standard timed automaton.

Example 2.23. Consider again the PTA $\mathscr{A}$ described in Example 2.22. Also consider the following reference valuation π of the parameters:

$$\pi: \; p_1 = 1 \;\; \wedge \;\; p_2 = 1$$

Then, the (non-parametric) timed automaton $\mathscr{A}[\pi]$ is the one described in Example 2.9. $\square$

We now define the notion of *acyclic* PTA. This acyclicity of a PTA can be deduced purely syntactically from its graphical representation, i.e., if this representation is acyclic.

Definition 2.24 (Acyclic PTA)**.** We say that a PTA is *graphically acyclic* (or, more simply, *acyclic*) if its associated graph is acyclic. $\blacksquare$

Note that the trace set associated with an acyclic PTA is necessarily acyclic.

Parallel Composition of PTAs. Similarly to the parallel composition of TAs (see Definition 2.10), we now introduce the notion of network of parametric timed automata, and show in the following definition how N PTAs can be composed into a single PTA.

Definition 2.25 (Network of PTAs)**.** Let $N \in \mathbb{N}$. For all $1 \leq i \leq N$, let $\mathscr{A}_i = (\Sigma_i, Q_i, (q_0)_i, X_i, P_i, K_i, I_i, \rightarrow_i)$ be a PTA. The sets Q_i, P_i, and X_i are mutually disjoint. A *network of parametric timed automata (NPTA)* is $\mathscr{A} = \mathscr{A}_1 \| \ldots \| \mathscr{A}_N$, where $\|$ is the operator for parallel composition defined in the following way. This NPTA corresponds to the PTA $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$ where

- $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,
- $Q = \Pi_{i=1}^{N} Q_i$,
- $q_0 = \langle (q_0)_1, \ldots, (q_0)_N \rangle$,
- $X = \biguplus_{i=1}^{N} X_i$,
- $P = \biguplus_{i=1}^{N} P_i$,
- $K = \bigwedge_{i=1}^{N} K_i$,
- $I(\langle q_1, \ldots, q_N \rangle) = \bigwedge_{i=1}^{N} I_i(q_i)$ for all $\langle q_1, \ldots, q_N \rangle \in Q$,

and $\rightarrow$ is defined as follows. For all $a \in \Sigma$, let T_a be the subset of indices $i \in 1,\ldots,N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $\langle q_1,\ldots,q_N\rangle \in Q$, for all $\langle q'_1,\ldots,q'_N\rangle \in Q$, $(\langle q_1,\ldots,q_N\rangle, g, a, \rho, \langle q'_1,\ldots,q'_N\rangle) \in \rightarrow$ if:

- for all $i \in T_a$, there exist g_i, ρ_i such that $(q_i, g_i, a, \rho_i, q'_i) \in \rightarrow_i$, $g = \bigwedge_{i \in T_a} g_i$, $\rho = \bigcup_{i \in T_a} \rho_i$, and,
- for all $i \notin T_a$, $q'_i = q_i$.

■

Example 2.26. We give in Figure 2.7 an example of Network of 2 PTAs. The left one (say, $\mathscr{A}'$) contains 3 locations (viz., q'_0, q'_1, and q'_2), 2 actions (viz., a and b), 1 clock x_1 and 1 parameter p_1. The right one (say, $\mathscr{A}''$) contains 3 locations (viz., q''_0, q''_2, and q''_3), 2 actions (viz., b and c), 1 clock x_2 and 1 parameter p_2.

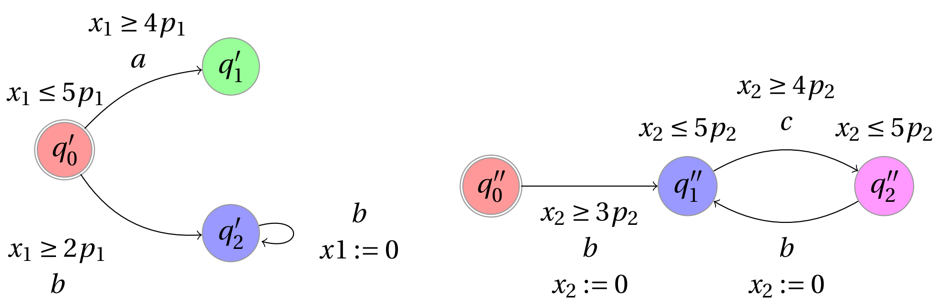


Figure 2.7: Example of Network of Parametric Timed Automata

Then, the composition of those 2 PTAs in parallel (viz., $\mathscr{A}' \| \mathscr{A}''$) corresponds to the PTA from Example 2.22, where $q_0 = (q'_0, q''_0)$, $q_1 = (q'_1, q''_0)$, $q_2 = (q'_2, q''_1)$ and $q_3 = (q'_2, q''_2)$.

□

Semantics

We now define the semantics of PTAs. We first introduce the notion of symbolic state.

Definition 2.27 (State). Let $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$ be a PTA. A *(symbolic) state s* of $\mathscr{A}(K)$ is a pair (q, C) where $q \in Q$ is a location, and $C \in \mathscr{K}_{X \cup P}$ a constraint on the clocks and the parameters. ■

For each valuation π of the parameters P, we may view a symbolic state s as the set of pairs (q, w) where w is a clock valuation such that $<w, \pi> \models C$.

In this thesis, we will be interested in checking whether the constraint associated with a symbolic state is satisfied by a given valuation of the parameters. This refers to the following notion of π-compatibility.

Definition 2.28 (π-compatibility)**.** Let $\mathscr{A}$ be a PTA, and $s = (q, C)$ be a state of $\mathscr{A}$. The state s is said to be *compatible with* π or, more simply, π*-compatible* if $\pi \models C$. ■

We now define the inclusion of a state in another one. This notion refers to the equality of definitions and inclusion of constraints, as formalized in the following definition.

Definition 2.29 (State inclusion)**.** We say that a state $s_1 = (q_1, C_1)$ is *included* in a state $s_2 = (q_2, C_2)$, denoted by $s_1 \subseteq s_2$, if $q_1 = q_2$ and $C_1 \subseteq C_2$.

We say that two states $s_1 = (q_1, C_1)$ and $s_2 = (q_2, C_2)$ are *equal*, denoted by $s_1 = s_2$, if $q_1 = q_2$ and $C_1 = C_2$. ■

We now define the inclusion of a set of states in another one. Observe that this notion does not refer to the inclusion of each state of the first set into a state of the second set, but to the *equality* of each state of the first set with a state of the second set.

Definition 2.30 (Set inclusion)**.** We say that a set of states S_1 is *included* into a set of states S_2, denoted by $S_1 \sqsubseteq S_2$, if

$$\forall s : s \in S_1 \Rightarrow s \in S_2.$$

We say that two sets of states S_1 and S_2 are *equal*, denoted by $S_1 = S_2$, if $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$. ■

The *initial state* of $\mathscr{A}(K)$ is a symbolic state s_0 of the form (q_0, C_0), where $C_0 = K \wedge I(q_0) \wedge \bigwedge_{i=1}^{H-1} x_i = x_{i+1}$. In this expression, K is the initial constraint on the parameters, $I(q_0)$ is the invariant of the initial state, and the rest of the expression lets clocks evolve from the same initial value.

The semantics of PTAs is given in the following under the form of a LTS.

Definition 2.31 (Semantics of PTAs)**.** Let $\mathscr{A} = (\Sigma, Q, q_0, X, P, K, I, \rightarrow)$ be a PTA. The *symbolic semantics of* $\mathscr{A}$ is the LTS $(S, S_0, \Rightarrow)$ over Σ where

$$S = \{(q, C) \in Q \times \mathscr{K}_{X \cup P} \mid C \subseteq I(q)\},$$
$$S_0 = \{(q_0, K \wedge I(q_0) \wedge \textstyle\bigwedge_{i=1}^{H-1} x_i = x_{i+1})\}$$

and the transition predicate $\Rightarrow$ is specified by the following three rules.

- $(q, C) \xrightarrow{a} (q', C')$ if $(q, g, a, \rho, q') \in \rightarrow$, and C' is a constraint on the clocks and parameters defined, using the set of (renamed) clock variables X', by:

  $C'(X') = (\exists X : (C(X) \wedge g(X) \wedge X' = \rho(X) \wedge I(q')(X')))$.

- $(q, C) \xrightarrow{d} (q, C')$, where d is a new parameter with values in $\mathbb{R}_{\geq 0}$, which means that C' is given by:

  $C'(X') = (\exists X : (C(X) \wedge X' = X + d \wedge I(q)(X')))$.

- $(q, C) \overset{a}{\Rightarrow} (q', C')$ if $\exists C''$ such that $(q, C) \xrightarrow{a} (q', C'')$ and $(q', C'') \xrightarrow{d} (q', C')$, i.e., C' is a constraint on the clocks and the parameters obtained by removing X and d from the following expression[1]:

  $C'(X') = (\exists X, d : (C(X) \wedge g(X) \wedge X' = \rho(X) \wedge I(q')(X') \wedge I(q')(X' + d)))$.

  It can be shown that C' can be put under the form of a (convex) constraint on the clocks and the parameters (using, e.g., Fourier-Motzkin elimination [Sch86]) of X and d.

■

Definition 2.32 (Symbolic run)**.** A step of the semantics of a PTA $\mathscr{A}(K)$ will be referred to as a *symbolic step* of $\mathscr{A}(K)$. Similarly, a run of the semantics of a PTA $\mathscr{A}(K)$ will be referred to as a *symbolic run* of $\mathscr{A}(K)$. ■

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

A symbolic run is represented by a directed graph where states are depicted within nodes containing the name of the location and the constraint on the clocks and the parameters, and transitions are depicted using edges labeled with the name of the action.

Example 2.33. Consider again the parametric timed automaton $\mathscr{A}$ of Example 2.22. Then Figure 2.8 depicts an example of symbolic run of $\mathscr{A}$.

□

We give below some results on the constraints on the parameters associated with the symbolic runs of a PTA, which will be used later in this thesis.

We first show that the constraint on the *parameters* associated with the symbolic states become more restrictive within a run, i.e., the constraint on the

[1]In $C'(X')$, we use the expression $I(q')(X') \wedge I(q')(X' + d)$ instead of $\forall 0 \leq e \leq d : I(q')(X' + e)$, using the fact that $I(q')$ is convex.

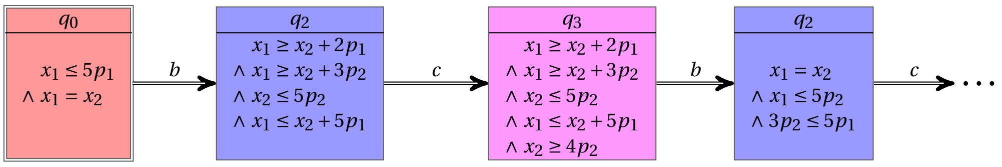


Figure 2.8: Example of symbolic run for a PTA

parameters of a given state of a run is included into the constraint on the parameters of a previous state of the same run.

Lemma 2.34. *Let $\mathscr{A}(K)$ be a PTA, and R a symbolic run of $\mathscr{A}$ of the form $(q_0, C_0) \overset{a_0}{\Rightarrow} \cdots (q_i, C_i) \overset{a_i}{\Rightarrow} (q_{i+1}, C_{i+1}) \overset{a_{i+1}}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (q_m, C_m)$. Then, we have:*

$$(\exists X : C_{i+1}) \subseteq (\exists X : C_i).$$

Proof. From the semantics of PTAs, C_{i+1} can be computed from C_i by addition of new constraints on the clocks and the parameters and elimination of clock variables only. Thus, C_{i+1} is more restrictive. ■

Note that the above result does not mean that the constraints on the *clocks and the parameters* become more restrictive within a run, due to the elimination of clock variables. The relation $C_{i+1} \subseteq C_i$ does not hold in the general case.

We now state that, given a PTA $\mathscr{A}(K)$, the constraint on the *parameters* associated with any symbolic state of $\mathscr{A}$ is included into K.

Lemma 2.35. *Let $\mathscr{A}(K)$ be a PTA, and (q, C) a symbolic state of a symbolic run of A. Then, we have:*

$$(\exists X : C) \subseteq K.$$

Proof. From the definition of the initial state (q_0, C_0) (see Definition 2.31), we have: $(\exists X : C_0) \subseteq K$. The result is then obtained by recurrence on Lemma 2.34. ■

Reachability and *Post* Computation

Recall from Definition 2.7 that a symbolic state s is reachable in one step from another symbolic state s' if s is the successor of s' in a symbolic run. This definition extends to sets of states. One defines $Post^i_{\mathscr{A}(K)}(S)$ as the set of states reachable from a set S of states in exactly i steps, and $Post^*_{\mathscr{A}(K)}(S)$ as the set of all states reachable from S in $\mathscr{A}(K)$ (i.e., $Post^*_{\mathscr{A}(K)}(S) = \bigcup_{i \geq 0} Post^i_{\mathscr{A}(K)}(S)$).

In this thesis, we will be in particular interested in computing the set $Post^*_{\mathscr{A}(K)}(\{s_0\})$, where s_0 is the initial state of $\mathscr{A}(K)$. Note that if $Post^{i+1}_{\mathscr{A}(K)}(\{s_0\}) = \emptyset$ (or, more generally, if $Post^{i+1}_{\mathscr{A}(K)}(\{s_0\}) \subseteq \bigcup_{j=0}^{i} Post^j_{\mathscr{A}(K)}(\{s_0\})$), then $Post^*_{\mathscr{A}(K)}(\{s_0\}) = \bigcup_{j=0}^{i} Post^j_{\mathscr{A}(K)}(\{s_0\})$.

Traces

The notion of trace associated with a concrete run, and the notion of trace set associated with a TA apply in a straightforward manner to PTAs.

Definition 2.36 (Trace associated with a symbolic run)**.** Given a PTA $\mathscr{A}$ and a symbolic run r of $\mathscr{A}$ of the form $(q_0, C_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (q_m, C_m)$, the *trace associated with* r is the alternating sequence of locations and actions $q_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} q_m$. We say that location q_i, for $1 \leq i \leq m$, *belongs* to the trace. ■

Note that the traces associated with symbolic runs of PTAs are the same mathematical object (i.e., alternating sequences of locations and actions) as the traces associated with concrete runs of TAs. As a consequence, we extend the notions of acyclicity and prefixes, defined for traces associated with concrete runs, to traces associated with symbolic runs. Moreover, we depict them under the same graphical form as traces associated with concrete runs, i.e., boxed nodes labeled with locations and double arrows labeled with actions.

Example 2.37. The trace associated with the symbolic run of Example 2.33 is depicted in Figure 2.9.

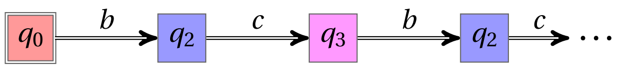


Figure 2.9: Example of trace associated with a symbolic run of a PTA

□

Definition 2.38 (Trace set)**.** Given a PTA $\mathscr{A}$, the *trace set of* $\mathscr{A}$ refers to the set of traces associated with the runs of $\mathscr{A}$. ■

As for the traces associated with concrete runs, we extend the notion of acyclicity of a trace to trace sets, and say that a trace set is *acyclic* if all its traces are acyclic.

Example 2.39. The trace set associated with the PTA of Example 2.22 is depicted in Figure 2.10.

□

Modeling Asynchronous Circuits Using PTAs

An application of TAs and PTAs is the modelization and the verification of asynchronous circuits. Throughout this thesis, we will model several asynchronous circuits using PTAs.

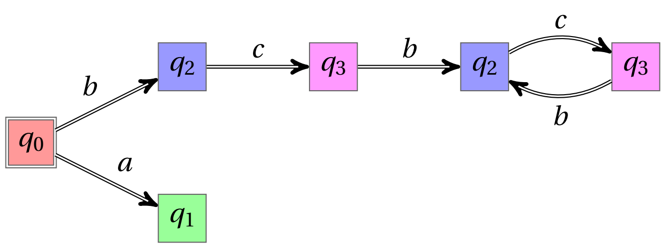


Figure 2.10: Example of trace set of a PTA

We consider a bi-bounded inertial model for gates (see [BS95, MP95]), where any change of the input may lead to a change of the output (after some delay). For a gate with n input signals, the delay associated with the change of the output signal of the gate depends on the configuration (low or high) of the n input signals, thus leading to 2^n different delays. Modeling those n possibilities would dramatically increase the complexity of the model. As a consequence, we usually use *intervals* of delays. Often, we consider only one interval for a gate, i.e., the minimum and the maximum of the possible delays associated with the 2^n input configurations. This option is usually a good compromise between precision of the result, and size of the model.

Another classical option is to consider two intervals, one corresponding to the rise of the output signal, the other corresponding to the fall of the output signal. This representation is interesting because delays corresponding to rise and fall can be very different. Although the size of the model is bigger than in the model with one interval of delays, this usually leads to a more precise state space containing less false executions, i.e., executions involving constraints on clocks and parameters which are actually not satisfiable.

Example 2.40. Consider the "NOT" gate depicted in Figure 2.11. This "NOT" gate has one input signal a, and one output signal b.

We first consider only one interval of delays. As a consequence, the interval of time between a change of the input and the change of the output is $[\delta^-;\delta^+]$, both for a rise and for a fall of the output.

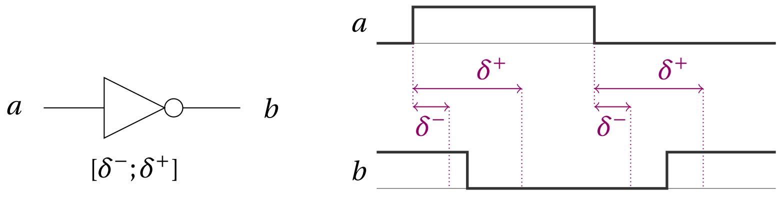


Figure 2.11: A "NOT" gate (left) and its environment (right)

Recall that, in stable mode, the output is equal to the inversion of the input. However, there may be configurations were both a and b are equal to each other, because of the delay between the change of the input and the change of the output. As a consequence, the PTA modeling this "NOT" gate contains four locations $n_{00}, n_{01}, n_{10}, n_{11}$, where n_{ij} stands for a configuration of the gate where the input a is equal to i and the output b is equal to j. We give in Figure 2.12 the PTA modeling this "NOT" gate, where the delay is modeled with an interval. This PTA contains one clock x and, as explained above, contains two parameters δ^- and δ^+, where $[\delta^-;\delta^+]$ represents the interval of time between a change of the input and the change of the output. Actions $a^\uparrow$ (resp. $a^\downarrow$) denotes the rise (resp. the fall) of signal a, and similarly for b.

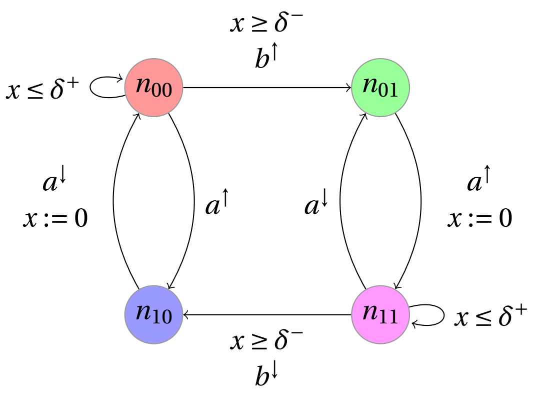


Figure 2.12: PTA modeling a "NOT" gate using one interval of delays

Let us first consider the stable configuration where a is equal to 0, and b is equal to 1, modeled by location n_{01}. When the input signal a changes (i.e., action $a^\uparrow$), the system enters an unstable configuration (location n_{11}), where it cannot stay more than δ^+ units of time: this is modeled by the reset of clock x on the transition labeled with $a^\uparrow$ from location n_{01} to location n_{11}, and the guard $x \leq \delta^+$ in location n_{11}. Moreover, the output b cannot change before δ^- units of time, which is modeled by the invariant $x \geq \delta^-$ associated with the transition labeled with $b^\downarrow$ from location n_{11} to location n_{10}. Finally note that, if the input a falls again while in location n_{11}, i.e., before the output could change, the system goes back to the initial stable configuration (location n_{01}) without any change of the output signal. The other stable configuration where a is equal to 1 and b is equal to 0, and modeled by location n_{10}, can be explained in a similar manner.

Let us now consider the model of this "NOT" gate using *two* intervals of delays. As a consequence, the interval of time between a change of the input and the change of the output is $[\delta^-_\uparrow;\delta^+_\uparrow]$ for a rise of the output, and $[\delta^-_\downarrow;\delta^+_\downarrow]$ for a fall of the output. We give the model using two interval of delays in Figure 2.13.

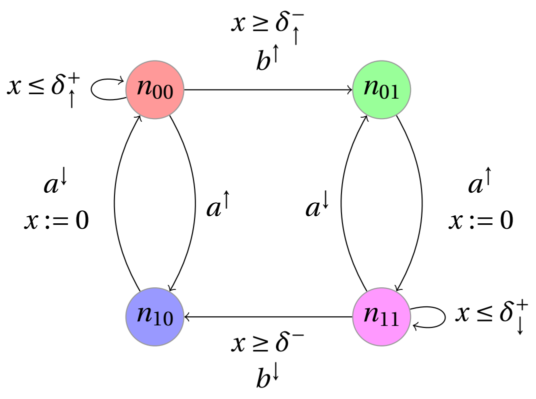


Figure 2.13: PTA modeling a "NOT" gate using a bi-bounded inertial model

Note that, in the case of this "NOT" gate, the number of locations and transitions remains the same as for the case with one interval (see Figure 2.12). Only the number of parameters gets multiplied by two. However, for systems with more that one input, the number of locations and transitions also increases when one switches from the model with one interval to the model with two intervals. □

2.3 The Good Parameters Problem

We now formally state the main problem we are interested to face in this thesis. As stated in Chapter 1, we are interested in finding *correct values* for the parameters of parametric timed automata. This synthesis of good parameters corresponds to the *good parameters problem*, as defined in [FJK08] in the framework of linear hybrid automata. We recall this problem below.

The Good Parameters Problem
Given a PTA $\mathscr{A}$ and a rectangular parameter domain $V_0 \subseteq \mathbb{R}_{\geq 0}^{M}$, what is the largest set of parameter values within V_0 for which $\mathscr{A}$ is safe?

As in [FJK08], we suppose that we are given a bounded rectangular parameter domain within which we want to synthesize good parameters. As a consequence, this problem could be referred to as a "bounded good parameters problem". However, for the sake of coherence with [FJK08], we will stick to the good parameter problems.

Note also that, as in [FJK08], we do not explicitly mention the property that makes the PTA $\mathscr{A}$ "safe". This only requirement is that this "safety" must be checked using the trace set of PTA. The fact that the main problem of this thesis does not mention the property is important, because we will see that some of the techniques we propose in this thesis do actually not depend on the property one wants to check.

2.4 Related Work

We discuss in this section several approaches to model distributed timed systems. We first justify our choice for the dense-time formalism in Section 2.4.1. We then recall Timed Automata in Section 2.4.2, and show some of the advantages of such a formalism. We present Time Petri Nets, and compare them to Timed Automata in Section 2.4.3. We also recall hybrid systems in Section 2.4.4, and compare them to Timed Automata. We finally justify in Section 2.4.5 our choice for the model of timed automata.

2.4.1 Representation of Time

When modeling timed systems, two major representations of time are used in the literature: the discrete time representation, and the dense (or continuous) time representation.

In the discrete time model, events can happen only at the integer time values. This allows the designer to describe the behavior of synchronous systems, where all components are driven by a single common global clock. This discrete time representation is the traditional model for synchronous hardware verification, where events (i.e., changes of signals) happen only at clock ticks, i.e., at the integer time values.

On the contrary, in the dense time representation, events can occur at any real (or rational) time value. As a consequence, models making use of a dense time representation are usually more complex than models using a discrete representation, but also more expressive.

In this thesis, we will focus on the dense time representation, and more specifically on the model of timed automata (see below). A classical formal justification for the dense time model can be found in [Alu92]. Furthermore, in this thesis, we will be in particular interested in verifying asynchronous circuits, where events (changes of signals) can happen at any real time value. As a consequence, the dense time representation is certainly more suitable than the discrete time representation. However, one could argue that, for a discrete time with enough precision (i.e., using time steps small enough), the discrete time representation can be suitable for the study of asynchronous circuits. Actually, it was shown that, in certain cases, finding the appropriate time step can be as difficult as doing the real-time model checking. The main interest of dense time representation, though, is that it gives a criterion of *robustness*. The discrete time representation can prove the correctness of a timed system for several integer values of the timing delays, but no information can be given inbetween two integers, whereas the dense time representation is able to guarantee *intervals* of values for which the system is correct. This is of particular interest in the sense that, when implementing a timed system, timing delays may slightly differ from the (exact and punctual) integer value they have been designed for.

2.4.2 Timed Automata

Timed Automata have been introduced in [AD94] as an extension of finite-state automata. Timed Automata allow the use of clocks, i.e., real-valued variables increasing linearly at the same rate (see Section 2.2.2).

Timed Automata have been used to model and successfully verify various case studies, e.g., communication protocols [DKRT97, CS01], and allowed famous bug discoveries [HSLL97]. It has been shown that model-checking for properties expressed the Timed CTL logic [ACD93] is decidable [ACD93, HNSY94] for timed automata and some of their extensions (e.g., [BDFP04]).

Recent work on Timed Automata focused on the notion of *robustness*, or implementability. As mentioned in [DDMR04], Timed Automata consider perfect clocks with infinite precision while implementations can only access time through finitely precise clocks. Moreover, Timed Automata react instantaneously to events while implementations can only react within a given usually small, but not zero, reaction delay. Also note that Timed Automata may describe control strategies that are unrealistic, like zeno-strategies or strategies that ask the controller to act faster and faster [CHR02]. As a consequence, models that have been proven correct may not be implementable. A first notion of robust timed automata has been considered in [GHJ97], with the following semantics: "if a robust timed automaton accepts a trajectory, then it must accept neighboring trajectories also and if a robust timed automaton rejects a trajectory, then it must reject neighboring trajectories also". The authors show in particular that the emptiness problem for robust timed automata is still decidable. Another work introducing the notion of "implementable" timed automata is given in [DDMR04]. This work is based on the "Almost ASAP semantics" (AASAP) defined in [DWDR05]. This semantics relaxes the classical semantics of timed automata in the sense that it does not impose a transition to be taken instantaneously but within a (very small) amount of time. The authors also show in [DDMR04] that the notion of robustness defined in [Pur00] is closely related to their notion of implementability. Robust model checking in this framework has been considered in [BMR06] with results of decidability.

The parametrization of Timed Automata into Parametric Timed Automata [AHV93], where parameters are used in guards and invariants in place of constants, allows parametric model checking. In other words, instead of checking if a given location can be reached, or if a given formula (expressed, e.g., using TCTL) is satisfiable, one can synthesize sets of values for the parameters under which the location can be reached (or under which the formula is satisfied). Unfortunately, most interesting problems related to PTAs have been shown to be undecidable for non-trivial PTAs. In particular, the emptiness problem is undecidable [AHV93] in the general case. However, decidability results are given in [HRSV02] for the verification of a special class, called "L/U automata", for which the emptiness problem is decidable. The authors also give a model-checking algorithm that uses parametric Difference Bound Matrices. Two subclasses of L/U automata, called lower-bound and upper-bound PTA, are also considered in [WY03], with decidability results. In [ZC05], the authors present a method to model check a real time system using parametric timed automata when a constraint over the parameter is given, i.e.: "given a real-time system and temporal formula, both of which may contain parameters, and a constraint over the parameters, does every allowed parameter assignment ensure that the real-time system satisfies the formula?" They then introduce an on-the-fly algorithm for solving this parametric model-checking problem. We also mention works related to the synthesis of parameters in the framework of PTAs in Section 3.6.

Various powerful model checkers allow to verify several classes of Timed Automata and their extensions (see Section 4.10 for a survey).

2.4.3 Time Petri Nets

Time Petri Nets [Mer74] are a classical and widely used extension of Petri Nets, allowing to model timed distributed systems using places, tokens and transitions that can be fired within a time interval. It has been shown that state reachability is decidable for bounded Time Petri Nets.

Both Time Petri Nets and Timed Automata are dense-time formalisms, which allow to study and verify dense-timed models, e.g., asynchronous circuits. As a consequence, their underlying state space is infinite, and verification techniques which enumerate exhaustively the state space cannot be applied. The main difference relies in the fact that, although both formalisms may be considered as infinite-state because of the real-valued time values, the number of locations in Timed Automata is *finite*, whereas Time Petri Nets remain an infinite marking model. Although the number of places of a Time Petri Net is bounded, the number of tokens in each node is (in the general case) unbounded, thus leading to a potentially infinite number of markings. Note however that several subclasses of Petri Nets consider a bounded number of tokens per place.

A further difference between Time Petri Nets and Timed Automata relies in the compositional approach. In Timed Automata, the compositional is purely syntactic and is defined by the formalism (the composition of Timed Automata is actually recalled in this thesis in Definition 2.10). On the contrary, composition of Time Petri Nets is less straightforward, because one has to specify the rules needed for composition within the Petri Net model.

Several classes of Time Petri Nets were shown to be equivalent to several classes of Timed Automata, and several approaches for translations from Time Petri Nets to Timed Automata have been proposed (see a survey in [PP06], as well as more recent work, e.g., [CR06, DDSS07, LRST09]).

A parametrization of Time Petri Nets with stopwatches (i.e., an extension of traditional clocks that can be suspended and resumed) has been considered in [TLR08], which allows to perform parametric model checking. As for Parametric Timed Automata, the parameters are used instead of constants in the firing conditions associated with the transitions. The authors propose semi-algorithms for the synthesis of parameter valuations satisfying a formula expressed using a subset of parametric TCTL formulae. Those algorithms have been implemented in the tool Roméo [LRST09], a software for Time Petri Nets analysis, making use of the Parma Polyhedra Library [BHZ08].

2.4.4 Hybrid Systems

Hybrid systems can be seen as a generalization of timed automata, where clocks may evolve at different rates. Actually, those "clocks" do not necessarily model the time elapsing, but can represent any real-valued continuous variable, such as temperature, speed, geographical position, etc. Hybrid automata were introduced in [ACHH92] and allow to define in any location a law of evolution with respect to time for each of the dynamic variables. A common subclass of hybrid systems consists in *linear hybrid systems*, where the derivatives of the variables are given within a (constant) interval for each location. An interesting survey on the different models of hybrid systems and linear hybrid automata, mostly in the 1990s, is given in [Fre05a] (Chapter 5).

The HyTech [HHWT95] model checker was one of the first tools allowing to model and verify hybrid systems. The tool PHAVer [Fre05b], which can be seen as a successor of HyTech, provides the user with a very efficient analysis of hybrid systems, using the Parma Polyhedra Library [BHZ08] (see Section 4.10 for a more complete survey on tools).

2.4.5 Discussion

Timed Automata provide the system designer with an intuitive and powerful way to verify distributed timed systems. Their compositional ability allow them to model distributed systems easily, using a single timed automaton for each component.

Moreover, they can be parameterized into Parametric Timed Automata [AHV93], allowing to synthesize parameter valuations corresponding to a given good behavior. The common drawbacks of Parametric Timed Automata rely in the state-space explosion problem, i.e., the quick saturation of the memory, and the undecidability of many standard problems [AHV93]. Concerning the former problem, we will introduce techniques allowing to avoid the state-space explosion problem. Concerning the latter, we will identify subclasses for which our techniques are decidable. When compared to hybrid automata, timed automata are less expressive, but also more efficient for to verification algorithms. The case studies we are interested in verifying in this thesis, viz., communication protocols and asynchronous circuits, can usually be modeled using standard timed automata, without the use of hybrid variables. Timed Automata (and their parametrization) can actually be seen as a formalism between Time Petri Nets and hybrid systems, in the sense that their composition can be seen as more intuitive than the composition of Time Petri Nets, and their formalism can be seen as less complex than hybrid automata.

Note that some of the results of this thesis apply to Time Petri Nets, when there exist translations from Timed Automata to Time Petri Nets. Extending the techniques introduced in this thesis to hybrid systems will be the subject of future work.

Chapter 3

An Inverse Method for Parametric Timed Automata

I'll sleep better knowing my good
friend is by my side to protect me.

The Good, the Bad and the Ugly
(Sergio Leone)

In Chapter 2, we introduced the good parameters problem, that aims at synthesizing values for timing parameters such that a system modeled by Parametric Timed Automata behaves well. In this chapter, we first consider the following inverse problem: "Given a reference valuation of the parameters, synthesize a constraint on the parameters such that, for any valuation satisfying this constraint, the trace set of the system is the same as under the reference valuation". This notion of equality of trace sets gives a guarantee of time-abstract equivalence of the behavior of the system. This problem can be seen as a subproblem of the good parameters problem for parametric timed automata, as defined in Section 2.3, and we will show in Chapter 5 how this inverse problem will allow us to solve the good parameters problem.

We introduce in this chapter a method solving the inverse problem. This *inverse method* supposes that we are given a parametric timed automaton $\mathscr{A}$ and a reference valuation π_0 of the parameters that one wants to generalize. The inverse method synthesizes a constraint K_0 on the parameters that corresponds to a set such that, for all valuations π of parameters in this set, the trace sets of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$ are equal, i.e., the behavior of the timed automaton $\mathscr{A}[\pi]$ is *(time-abstract) equivalent* to the behavior of $\mathscr{A}[\pi_0]$.

This method has three main advantages. First, it gives a criterion of *robustness* by ensuring the correctness of the system for other values for the parameters around the reference valuation. This is of interest when implementing a system: indeed, the exact model with (for example) integer values for timing delays that has been formally verified will necessarily be implemented using values which will not be exactly the ones that have been verified. Second, it allows the system designer to *optimize* some delays without changing the overall functional behavior of the system. Third, it allows us to *rescale* the constants of the system, allowing to verify the system (e.g., using an external model checker) with much smaller constants, often leading to a high decrease of the verification time and the state space.

Plan of the Chapter. We first formally define the inverse problem in Section 3.1 by using the example of "flip-flop" asynchronous circuit considered in the introduction. We then introduce the inverse method in Section 3.2. We then give in Section 3.3 results of correctness and termination, and state properties of the method. We also show the advantages of the inverse method. We apply our inverse method to the motivating example in Section 3.4. In Section 3.5, we present variants of the inverse method, solving different problems from the inverse problem, and discuss the applications. We finally present work related to the synthesis of parameters for timed systems in Section 3.6.

3.1 The Inverse Problem

3.1.1 A Motivating Example

Consider again the example of flip-flop circuit, introduced in Section 1.1. Let us model this circuit using the formalism of Parametric Timed Automata described in Chapter 2.

Each gate is modeled by a PTA, as well as the environment. Note that element G_4 is a "NOT" gate; recall that an example of PTA modeling such a "NOT" gate is given in Figure 2.12 page 33. The PTA $\mathscr{A}$ modeling the system results from the composition of those 5 PTAs. Each location of $\mathscr{A}$ corresponds to a different value of the signals D, CK, g_1, g_2, g_3 and g_4 (recall that $g_4 = Q$).

The initial location q_0 corresponds to the initial levels of the signals according to the environment. Recall that we consider an environment starting from $D = CK = Q = 0$ and $g_1 = g_2 = g_3 = 1$, with the following ordered sequence of actions for inputs D and CK: $D^{\uparrow}$, $CK^{\uparrow}$, $D^{\downarrow}$, $CK^{\downarrow}$, as depicted in Figure 1.1 right. Therefore, we have the implicit constraint $T_{Setup} \leq T_{LO} \wedge T_{Hold} \leq T_{HI}$. The initial constraint C_0 (regardless of the equality between the clock variables, see Section 2.2.3) is:

$$T_{Setup} \leq T_{LO} \wedge T_{Hold} \leq T_{HI} \wedge \bigwedge_{i=1,..,4} \delta_i^- \leq \delta_i^+$$

We consider that the circuit has a *good behavior* if it verifies the following property $Prop_1$: "every trace contains both $Q^\uparrow$ and $CK^\downarrow$, and $Q^\uparrow$ occurs before $CK^\downarrow$".

We consider the following valuation π_0 of the parameters[1]:

$$\begin{array}{llll} T_{HI} = 24 & T_{LO} = 15 & T_{Setup} = 10 & T_{Hold} = 17 \\ \delta_1^- = 7 & \delta_1^+ = 7 & \delta_2^- = 5 & \delta_2^+ = 6 \\ \delta_3^- = 8 & \delta_3^+ = 10 & \delta_4^- = 3 & \delta_4^+ = 7 \end{array}$$

Let us study the behavior of the flip-flop circuit under π_0. The trace set of $\mathscr{A}[\pi_0]$ is depicted in Figure 3.1, where the meaning of each location in terms of signals is given in Table 3.1. Recall that we do not depict each trace separately, but depict the trace set under the form of a tree or a graph. However, this graph structure is only used for the sake of simplicity of representation of the possible traces, and does not contain any information on the possible branching behavior of the system.

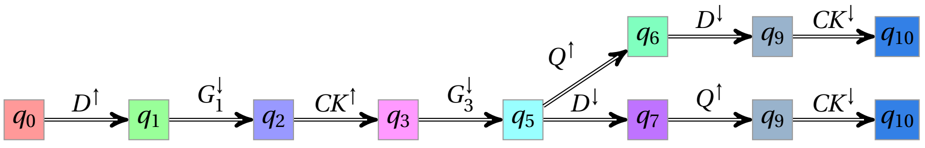

$$\begin{tikzcd} & & & & & q_6 \arrow[r, "D^\downarrow"] & q_9 \arrow[r, "CK^\downarrow"] & q_{10} \\ q_0 \arrow[r, "D^\uparrow"] & q_1 \arrow[r, "G_1^\downarrow"] & q_2 \arrow[r, "CK^\uparrow"] & q_3 \arrow[r, "G_3^\downarrow"] & q_5 \arrow[ur, "Q^\uparrow"] \arrow[r, "D^\downarrow"] & q_7 \arrow[r, "Q^\uparrow"] & q_9 \arrow[r, "CK^\downarrow"] & q_{10} \end{tikzcd}$$

Figure 3.1: Trace set of the flip-flop circuit under π_0

Location	D	CK	g_1	g_2	g_3	g_4
q_0	0	0	1	1	1	0
q_1	1	0	1	1	1	0
q_2	1	0	0	1	1	0
q_3	1	1	0	1	1	0
q_4	0	1	0	1	1	0
q_5	1	1	0	1	0	0
q_6	1	1	0	1	0	1
q_7	0	1	0	1	0	0
q_8	0	0	0	1	0	0
q_9	0	1	0	1	0	1
q_{10}	0	0	0	1	0	1

Table 3.1: Locations of the flip-flop circuit

Each of the two traces depicted in this trace set contains both $Q^\uparrow$ and $CK^\downarrow$, and $Q^\uparrow$ occurs before $CK^\downarrow$. As a consequence, this trace set is a *good* trace set according to the property we want to verify.

[1]This valuation was actually synthesized in order to satisfy the constraint guaranteeing a good behavior and given in [CC07]. We give this constraint Z in Section 3.4.

We are now interested in studying the evolution of the behavior of the system if one changes some of the values of the parameters. More precisely, we are interested in identifying parameter valuations for which the system has exactly the same (good) behavior, i.e., exactly the same trace set.

3.1.2 The Problem

More generally, the *inverse problem* can be stated as follows [ACEF09]:

The Inverse Problem
Given a PTA $\mathscr{A}$ and a reference valuation π_0, find a constraint K_0 on the parameters such that:

- $\pi_0 \models K_0$, and
- for all $\pi \models K_0$, the trace sets of $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$ are the same.

This problem considers an equality of trace sets between $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$, and thus guarantees a *time-abstract equivalence* between the behavior of $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$. This problem is a subproblem of the good parameters problem, as introduced in Section 3.1, in the sense that the inverse problem focuses on the synthesis of parameters for which the system has the *same* behavior as under a given reference valuation, whereas the good parameters problem focuses on the synthesis of *all* the parameter valuations (within a finite parameter subset) corresponding to (possibly different) good behaviors. We will see in Chapter 5 how the inverse method solving this inverse problem can be used to solve the good parameters problem.

3.2 The Inverse Method Algorithm

3.2.1 Principle

We introduce in the following the *inverse method* [ACEF09], which is a solution to the inverse problem stated above. The inverse method consists in generating runs starting from the initial state, and removing states incompatible with the reference values by appropriately refining the current constraint K_0 on the parameters. The generation procedure is then restarted until a new incompatible state is produced, and so on, iteratively until no incompatible state is generated.

We first informally describe the algorithm *IM* in the following. Starting with $K = \texttt{true}$, we iteratively compute a growing set of reachable states. When a π-*incompatible* state (q, C) is encountered (i.e., when $\pi \not\models C$), K is refined as follows: a π-incompatible inequality J (i.e., such that $\pi \not\models J$) is selected within the projection of C onto the parameters and $\neg J$ is added to K. The procedure is then started again with this new K, and so on, until no new state is computed. We finally return the intersection of the projection onto the parameters of all the constraints associated with the reachable states.

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

```
input : A PTA $\mathcal{A}$ of initial state $s_0 = (q_0, C_0)$
input : Valuation $\pi$ of the parameters
output: Constraint $K_0$ on the parameters

$i \leftarrow 0$; $K \leftarrow \texttt{true}$; $S \leftarrow \{s_0\}$
while true do
    while there are $\pi$-incompatible states in $S$ do
        Select a $\pi$-incompatible state $(q, C)$ of $S$ (i.e., s.t. $\pi \not\models C$) ;
        Select a $\pi$-incompatible $J$ in $(\exists X : C)$ (i.e., s.t. $\pi \not\models J$) ;
        $K \leftarrow K \wedge \neg J$ ;
        $S \leftarrow \bigcup_{j=0}^{i} Post_{\mathcal{A}(K)}^{j}(\{s_0\})$ ;
    if $Post_{\mathcal{A}(K)}(S) \sqsubseteq S$ then
        return $\bigcap_{(q,C) \in S} (\exists X : C)$
    $i \leftarrow i + 1$ ;
    $S \leftarrow S \cup Post_{\mathcal{A}(K)}(S)$ ;        // $S = \bigcup_{j=0}^{i} Post_{\mathcal{A}(K)}^{j}(\{s_0\})$
```

We now give the inverse method algorithm $IM(\mathcal{A}, \pi)$ in Algorithm 1. The algorithm is made of two **do** loops. The inner **do** loop removes all the π-incompatible states for a given depth i of the runs, i.e., states of $Post_{\mathcal{A}(K)}^{i}(\{s_0\})$. This removal is made by removing a π_0-incompatible inequality randomly selected within the projection onto the parameters of the constraint C associated with a π_0-incompatible state; this projection onto the parameters, i.e., elimination of the clocks, is denoted by $\exists X : C$ in the algorithm. The outer **do** loop iteratively computes the set of all reachable states. When the fixpoint is reached, i.e., when all new states computed at iteration i are equal to states computed at previous iterations ($Post_{\mathcal{A}(K)}(S) \sqsubseteq S$), the intersection K_0 of all the constraints on the parameters associated with the states of S is returned.

Actually, the two major steps of the algorithm are the following ones:

1. the iterative negation of the π-incompatible states (by negating a π-incompatible inequality J) prevents for any $\pi' \models K_0$ the behavior different from π;

2. the intersection of the constraints associated with all the reachable states guarantees that all the behaviors under π are allowed for all $\pi' \models K_0$.

3.2.2 A Toy Example

Let us consider the PTA given in Figure 3.2, following the formalism defined in Chapter 2. This PTA contains two clocks x_1 and x_2, three parameters p_1, p_2 and p_3, and three locations q_0, q_1 and q_2. The initial location q_0 has invariant $x_1 \leq p_1$. The transition from q_0 to q_1, labeled a, has guard $x_2 \geq p_2$, and resets x_1. The transition from q_0 to q_2, labeled b, has guard $x_1 \geq p_3$, and does not reset any clock.

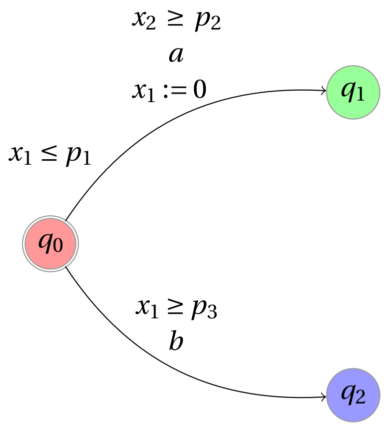


Figure 3.2: A toy PTA

Let us assume that q_2 corresponds to a “bad location”. Classical methods, using this information, will synthesize the constraint $Z : p_1 < p_3$, which guarantees that the location is not reachable. Suppose now that we are given the following “good” valuation of the parameters $\pi_0 : p_1 = 4 \wedge p_2 = 2 \wedge p_3 = 6$, under which the PTA is assumed to have a “good” behavior. Then our inverse method will synthesize the constraint $K_0 : p_1 < p_3 \wedge p_2 \leq p_1$. For all valuations π of the parameters satisfying K_0, our method guarantees that the PTA behaves in the same manner as under π_0. We are thus ensured that the behavior of the PTA is correct. We can note that K_0 is strictly smaller than Z. On the one hand, this may be viewed as a limitation of our method. On the other hand, this may indicate that there are incorrect behaviors other than those corresponding to the reaching of q_2. For example, there are some parameter valuations satisfying Z, under which a deadlock of the PTA occurs at the initial location q_0. In contrast, our inverse method guarantees that such a deadlock is impossible under any instance satisfying K_0 (because the deadlock does not occur under π_0).

3.2.3 Remarks on the Algorithm

We explain in the following some of our choices.

Fixpoint. As shown in Line 8 in Algorithm 1, the algorithm stops when $Post_{\mathcal{A}(K)}(S) \sqsubseteq S$, i.e., when each of the new states encountered at step i is *equal* to a previously encountered state (see Definition 2.30). A more standard solution would be to consider an inclusion of the states, i.e., each of the new states encountered at step i is *included* in a previously encountered state (recall that state inclusion means equality of locations and inclusion of constraints, see Definition 2.29). More formally, this more standard fixpoint condition would be:

$$\forall s \in Post_{\mathcal{A}(K)}(S) \exists s' \in S : s \subseteq s'.$$

However, this condition is not sufficient, because we do not guarantee the equality of traces in that case.

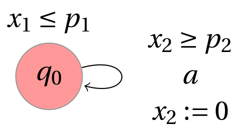


Figure 3.3: PTA showing the necessity of the fixpoint of *IM*

Consider the PTA $\mathcal{A}$ depicted in Figure 3.3, containing 2 clocks x_1 and x_2, two parameters p_1 and p_2, and a single state q_0 with invariant $x_1 \leq p_1$. The only transition is a self-loop through action a from q_0, with guard $x_2 \geq p_2$ and resetting x_2. Now consider the following reference valuation π_0 of the parameters: $p_1 = 3 \wedge p_2 = 1$. The trace set of $\mathcal{A}[\pi_0]$, depicted in Figure 3.4, is made of three self-transitions from q_0.

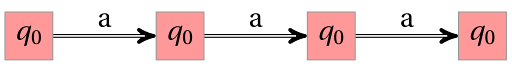


Figure 3.4: Trace set of $\mathcal{A}[\pi_0]$

Let us apply the inverse method to $\mathcal{A}$ and π_0. Consider for the sake of simplicity that we have initially $x_1 = x_2 = 0$. Then the initial state after time-elapsing is (q_0, C_0), with $C_0 : x_2 = x_1 \wedge x_1 \geq 0 \wedge p_1 \geq x_1$. After one iteration of *IM*, the state (q_0, C_1) is reachable, with $C_1 : x_1 \geq x_2 + p_2 \wedge x_2 \geq 0 \wedge x_1 \geq x_2 \wedge p_1 \geq x_1$. The projection of C_1 onto the parameters is $p_1 \geq p_2$, which is π_0-compatible. Since C_1 is obviously not included into C_0, the algorithm goes further. We then reach state (q_0, C_2), with $C_1 : x_1 \geq x_2 + 2p_2 \wedge x_2 \geq 0 \wedge x_1 \geq x_2 \wedge p_1 \geq x_1$. The projection of C_2 onto the parameters is $p_1 \geq 2p_2$, which is π_0-compatible. Now, we have $C_2 \subseteq C_1$. So this more standard fixpoint condition given above is satisfied, and the algorithm would return the intersection of the projection onto the parameters of the constraints associated with the 2 states, i.e., $p_1 \geq p_2$. The problem is that, under this constraint, say K'_0, the trace set is the one depicted in Figure 3.5, which is not equal to the trace set of $\mathscr{A}[\pi_0]$ (depicted in Figure 3.4).

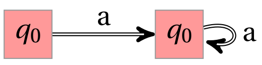


Figure 3.5: Trace set of $\mathscr{A}[\pi]$, for any $\pi \models K'_0$

Note that the correct constraint K_0 guaranteeing the same trace set as $\mathscr{A}[\pi_0]$, and output by our algorithm *IM*, is actually: $4p_2 > p_1 \wedge p_1 \geq 3p_2$.

This variant of the algorithm may actually still be of interest when one wants to consider the equality of the traces of $\mathscr{A}[\pi_0]$ and the traces of $\mathscr{A}[\pi]$ up to some length, for any $\pi \models IM(\mathscr{A}, \pi_0)$, or when one is only interested in non-reachability properties. This will be formalized in the variant $IM^{\subseteq}$ of the inverse method, detailed in Section 3.5.1.

Final intersection. The reason why the intersection of the constraints associated with all the reachable states is returned (see line 9 in Algorithm 1), and not the current constraint K as one could have expected, comes from the necessity to avoid deadlocks which do not occur in $\mathscr{A}[\pi_0]$. Consider the PTA $\mathscr{A}$ depicted in Figure 3.6, containing one clock x and two parameters p_1 and p_2, and the following reference valuation of the parameters π_0: $p_1 = 2 \wedge p_2 = 1$. It is easy to see that the trace set of $\mathscr{A}[\pi_0]$ corresponds to the only trace: $q_0 \overset{a}{\Rightarrow} q_1$.

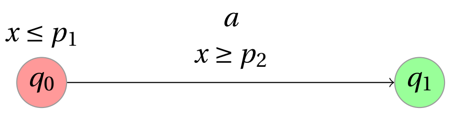


Figure 3.6: PTA explaining the intersection of constraints returned by *IM*

When applying the inverse method algorithm to $\mathscr{A}$ and π_0, one can see that the constraint on the parameters associated with q_0 is `true`, and the constraint associated with q_1 is $p_2 \leq p_1$, which are both obviously π_0-compatible. As a consequence, we have that $K = \texttt{true}$ at the end of *IM*. Consider now the following valuation π_1: $p_1 = 1 \wedge p_2 = 2$. It is easy to see that the trace set of $\mathscr{A}[\pi_1]$ corresponds to the only trace q_0, i.e., only the initial location, because the constraint associated with q_1 is not satisfied by π_1. As a consequence, the trace sets of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi_1]$ are different, although $\pi_1 \models K$. By adding to K the intersection of the constraint associated with all the reachable states, we have the guarantee that all the transitions will be fired, thus avoiding the deadlocks which do not occur in $\mathscr{A}[\pi_0]$. Note that this intersection is included in K (see Lemma 3.1), which is the reason why only the intersection is returned. Also note that this intersection is necessarily satisfiable, because all the constraints associated with a reachable state are π_0-compatible (see Lemma 3.2, and paragraph below).

Interest of the Reference Valuation. The test of π_0-compatibility, i.e., the comparison to a reference valuation of the parameters, is essential in the algorithm, and is the reason why the constraint output is necessarily satisfiable. Suppose that the inverse method was based on a *reference trace set* (or, more simply, a reference trace) instead of a reference valuation: when encountering a bad state (i.e., a state which would not belong to the reference trace), one could negate any inequality of the constraint associated with this state. As a consequence, we would have no guarantee that the constraint K is satisfiable. In our inverse method, only the test of π_0-compatibility guarantees that the selected inequality is π_0-incompatible (and thus its negation is π_0-compatible), which guarantees that the constraint K is necessarily π_0-compatible, and thus satisfiable.

We also investigated variants of the algorithm with *two* reference valuations (say, π_1 and π_2), but this case is much more tricky than with a single reference valuation, and does not seem to bring interesting results. Indeed, if one removes inequalities incompatible *either* with π_1 or with π_2, then the constraint K soon gets insatisfiable. If one removes inequalities incompatible *both* with π_1 and with π_2, then one may keep bad behaviors (depending with our definition of bad behavior). And if one removes all π_1-incompatible inequalities, but try to select only π_2-incompatible inequalities within the π_1-incompatible inequalities (unless there is no other choice), then in practice we eventually still have to negate at least one π_2-compatible inequality, which lets K become π_2-incompatible, and does not bring anything more than our algorithm *IM* applied to π_1 only.

Nondeterminism. Note that there are two possible sources of nondeterminism in the algorithm:

1. when one selects a π_0-incompatible state (q, C) (i.e, $\pi_0 \not\models \exists X : C$), and

2. when one selects an inequality J among the conjunction of inequalities $\exists X : C$, that is "responsible" for this π_0-incompatibility (i.e., such that $\pi_0 \not\models J$, hence $\pi_0 \models \neg J$).

This nondeterminism is the reason why the algorithm *IM* is non-confluent, i.e., several applications of *IM* to the same input may lead to the output of different constraints (see Proposition 3.12).

3.3 Results

3.3.1 Correctness

We now formally establish the correctness of Algorithm *IM*.

We suppose in this subsection that $IM(\mathscr{A}, \pi_0)$ terminates with output K_0. Let K (resp. S) be the current constraint on the parameters (resp. the current set of reachable states) when the algorithm terminates. We have $S = Post^*_{\mathscr{A}(K)}(\{s_0\})$ and $K_0 = \bigcap_{(q,C)\in S}(\exists X : C)$.

Lemma 3.1. *We have $K_0 \subseteq K$.*

Proof. From Lemma 2.35, for all states $(q, C) \in S$, we have $(\exists X : C) \subseteq K$, since $S = Post^*_{\mathscr{A}(K)}(\{s_0\})$. As $K_0 = \bigcap_{(q,C)\in S}(\exists X : C)$, then $K_0 \subseteq K$. ■

Let us now show that $\pi_0 \models K_0$. Formally:

Lemma 3.2. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. We have: $\pi_0 \models K_0$.*

Proof. When Algorithm *IM* terminates, the set S is π_0-compatible (i.e., $\pi_0 \models (\exists X : C)$, for all $(q, C) \in S$). Thus, the intersection K_0 of the constraints associated with the states of S, i.e., $\bigcap_{(q,C)\in S}(\exists X : C)$, is satisfied by π_0. ■

Let us now show that the set of traces in the concrete semantics and the set of traces in the symbolic semantics are equal. This will lead to Theorem 3.8, stating the correctness of Algorithm *IM*.

First of all, we state that, for all $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$, we can find an equivalent concrete run of $\mathscr{A}[\pi]$.

Lemma 3.3. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. For all π such that $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$ reaching (q, C), there exists a clock valuation w such that $<w, \pi> \models C$.*

Proof. For each symbolic run of $\mathscr{A}(K)$ reaching (q, C), we have $(q, C) \in S$ since $S = Post^*_{\mathscr{A}(K)}(\{s_0\})$. Moreover, we have $K_0 = \bigcap_{(q,C)\in S}(\exists X : C)$. Thus, for all $\pi \models K_0$, for all $(q, C) \in S$, we have $\pi \models (\exists X : C)$. Hence, there exists a clock valuation w such that $<w, \pi> \models C$. ■

Lemma 3.4. *Let $\mathscr{A}(K)$ be a PTA. For each symbolic run of $\mathscr{A}(K)$ reaching (q, C), for each parameter valuation π and clock valuation w such that $<w,\pi> \models C$, there exists an equivalent concrete run of $\mathscr{A}[\pi]$ reaching (q, w).*

Proof. The proof of Proposition 3.17 in [HRSV02] can be adapted in a straightforward manner. ■

Proposition 3.5. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. For all $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$, there exists an equivalent concrete run of $\mathscr{A}[\pi]$.*

Proof. From Lemma 3.3 and Lemma 3.4. ■

Conversely, we now state that, for all $\pi \models K_0$, for each concrete run of $\mathscr{A}[\pi]$, we can find an equivalent symbolic run of $\mathscr{A}(K)$.

Proposition 3.6. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. For all $\pi \models K_0$, for each concrete run of $\mathscr{A}[\pi]$, there exists an equivalent symbolic run of $\mathscr{A}(K)$.*

Proof. The proof of Proposition 3.18 in [HRSV02] can be adapted in a straightforward manner to show that, for all $\pi \models K$, for each concrete run of $\mathscr{A}[\pi]$, there exists an equivalent symbolic run of $\mathscr{A}(K)$. The result follows from the fact that $\pi \models K_0$ implies $\pi \models K$ (by Lemma 3.1). ■

We can thus now state the equivalence of trace sets.

Proposition 3.7. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. For all $\pi \models K_0$, the sets of runs of $\mathscr{A}(K)$ and $\mathscr{A}[\pi]$ are equivalent, i.e., the trace sets are equal.*

Proof. From Proposition 3.5 and Proposition 3.6. ■

The following theorem formally states the correctness of our inverse method algorithm, and shows that it solves the inverse problem as defined in Section 3.1.2.

Theorem 3.8 (Correctness of *IM*)**.** *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Suppose that $IM(\mathscr{A}, \pi_0)$ terminates with output K_0. Then, we have:*

1. *$\pi_0 \models K_0$, and*
2. *for all $\pi \models K_0$, the sets of concrete runs of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$ are equivalent, i.e., the trace sets are equal.*

Proof. From Lemma 3.2 and Proposition 3.7. ■

3.3.2 Termination

Reachability analysis is known to be undecidable in the framework of PTAs [AHV93, Doy07], and computations performed with tools on PTAs (such as HyTech [HHWT95]) do not always terminate. However, we give a sufficient condition for ensuring termination of our method.

We first show that, if all traces of $\mathscr{A}[\pi_0]$ are finite, i.e., if there exists $n \in \mathbb{N}$ s.t. $Post^n_{\mathscr{A}[\pi_0]}(\{s_0\}) = \emptyset$, then the algorithm terminates.

Proposition 3.9 (Termination in the acyclic case)**.** *Let $\mathscr{A}$ be a PTA and π_0 be a valuation of P. If there exists $n \in \mathbb{N}$ s.t. $Post^n_{\mathscr{A}[\pi_0]}(\{s_0\}) = \emptyset$, then algorithm $IM(\mathscr{A}, \pi_0)$ terminates in at most n iterations of the outer* ***do*** *loop.*

Proof. Let us first consider the inner **do** loop for a given i. At each iteration, we select a state $s = (q, C)$ of $Post^i_{\mathscr{A}(K)}(\{s_0\})$. We select an inequality J in $\exists X : C$, negate J, and add it to K. Hence, s does not belong to $Post^i_{\mathscr{A}(K \wedge \neg J)}(\{s_0\})$. The traces of $\mathscr{A}(K)$ of length $j \leq i$ can be organized under the form a finite tree, say T. Likewise, the traces of $\mathscr{A}(K \wedge \neg J)$ of length $j \leq i$ can be organized under the form a finite tree, say T'. It is easy to show that T' is a subtree of T, i.e., each branch starting from the root of T' is a (sub)branch starting from the (same) root in T. Thus no new state can be reached in $\mathscr{A}(K \wedge \neg J)$. Moreover, the branch of T reaching the location corresponding to s does not belong to T'. So, the number of nodes of T' is less than the number of nodes of T. Hence, the number of states of $Post^i_{\mathscr{A}(K \wedge \neg J)}(\{s_0\})$ is less than the number of states of $Post^i_{\mathscr{A}(K)}(\{s_0\})$. Thus, the inner **do** loop terminates.

Let us now consider the outer **do** loop. Since $Post^n_{\mathscr{A}[\pi_0]}(\{s_0\}) = \emptyset$, the concrete runs of $\mathscr{A}[\pi_0]$ have at most length $n-1$. Let us show by *reductio ad absurdum* that the outer **do** loop terminates at iteration $i \leq n-1$. Suppose that we are still in the outer **do** loop at $i = n$. Thus, there exists a symbolic run of $\mathscr{A}(K)$ of length n of the form $(q_0, C_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{n-2}}{\Rightarrow} (q_{n-1}, C_{n-1}) \overset{a_{n-1}}{\Rightarrow} (q_n, C_n)$. Moreover, since all the states in $S = Post^i_{\mathscr{A}(K)}(\{s_0\})$ are π_0-compatible, we have $\pi_0 \models C_i$, for $0 \leq i \leq n$. Hence, there exists w such that $<w, \pi_0> \models C_n$. Therefore, by Lemma 3.4, there exists an equivalent concrete run of length n of $\mathscr{A}[\pi_0]$ reaching (q_n, w), which contradicts the assumption $Post^n_{\mathscr{A}[\pi_0]}(\{s_0\}) = \emptyset$. ■

A sufficient (but non necessary) condition so that there exists $n \in \mathbb{N}$ such that $Post^n_{\mathscr{A}[\pi_0]}(\{s_0\}) = \emptyset$ is that the trace set of $\mathscr{A}[\pi_0]$ be acyclic, i.e., the oriented graph depicting the trace set be acyclic. Recall from Definition 2.16 that, in that case, traces never pass twice by the same location. This is generally the case for synchronous circuits analyzed over a fixed number (typically, 1 or 2) of clock cycles.

A sufficient (but non necessary) condition for the acyclicity of the trace set of $\mathscr{A}$ is that $\mathscr{A}$ be itself acyclic (see Definition 2.24).

Now, notice that the fixpoint condition given at line 8 of Algorithm 1 is not that there exists $n \in \mathbb{N}$ such that $Post^n_{\mathscr{A}(K)}(\{s_0\}) = \emptyset$, but that there exists $n \in \mathbb{N}$ such that all the states computed at iteration n are included into states computed at previous iterations, i.e., $Post^n_{\mathscr{A}(K)}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(K)}(\{s_0\})$. Considering the fixpoint condition, it is straightforward to show that, if there exists $n \in \mathbb{N}$ such that, for all K, $Post^n_{\mathscr{A}(K)}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(K)}(\{s_0\})$, then Algorithm *IM* terminates. In such a case, the oriented graph depicting the trace set of $\mathscr{A}[\pi_0]$ is not necessarily acyclic and may contain loops.

Theorem 3.10 (Termination in the cyclic case)**.** *Let $\mathscr{A}$ be a PTA and π_0 be a valuation of P. If there exists $n \in \mathbb{N}$ such that $Post^n_{\mathscr{A}(\texttt{true})}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(\texttt{true})}(\{s_0\})$, then algorithm IM($\mathscr{A},\pi_0$) terminates.*

Proof. From the semantics of PTAs, given K a constraint on the parameters synthesized during our algorithm, the set of runs of $\mathscr{A}(K)$ is more restricted than the set of runs of $\mathscr{A}(\texttt{true})$: less states may be reachable, and the constraints associated with the states are more restrictive (in the sense of constraint inclusion). This comes from the finiteness of the number of inequalities possibly synthesized by our algorithm, due to the finiteness of the symbolic structure (set of symbolic cyclic runs) representing the semantics of $\mathscr{A}(\texttt{true})$. As a consequence, if there exists $n \in \mathbb{N}$ such that $Post^n_{\mathscr{A}(\texttt{true})}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(\texttt{true})}(\{s_0\})$, then we have $Post^n_{\mathscr{A}(K)}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(K)}(\{s_0\})$, for any K. Finally, from the fixpoint of Algorithm *IM*, it is straightforward to see that, if there exists $n \in \mathbb{N}$ such that, for all K, $Post^n_{\mathscr{A}(K)}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(K)}(\{s_0\})$, then Algorithm *IM* terminates. ■

Although the cyclicity of *runs* of $\mathscr{A}(K)$ is a sufficient condition of termination of *IM*, it is important to point out that the cyclicity of *traces* of $\mathscr{A}(K)$ is not a sufficient condition for the termination. Indeed, recall that traces are time-abstract runs, and cyclic traces may refer to diverging runs, where the constraints on the clocks and the parameters associated with the locations are incomparable.

For most of the case studies the inverse method was applied to, termination was ensured. However, it is possible to find examples for which the inverse method does not terminate.

Example 3.11. Consider the PTA depicted in Figure 3.7, which contains one location, and two clocks x_1 and x_2 (although x_1 does not appear on the graph depicting the PTA). Consider the reference valuation $\pi_0 : p_1 = 1$.

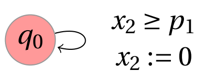


Figure 3.7: An example of PTA for which *IM* does not terminate

The application of the inverse method algorithm to this PTA and π_0 does not terminate, because the algorithm will generate an infinite sequence of states with constraints of the form $x_1 \geq x_2 + i * p_1$, with i increasing. Actually, the application of the inverse method algorithm to this PTA does not terminate for any reference valuation of the parameter. □

Recall that a decidable subclass of PTAs is introduced in [HRSV02], namely L/U automata. For this subclass of PTAs, our algorithm *IM* may also not terminate. Actually, the PTA considered in Example 3.11 above and for which *IM* does not terminate falls into this class of L/U automata, because the only parameter p_1 only appears as an upper bound.

3.3.3 Properties

Non-confluence. We first show that the algorithm *IM* is non-confluent: for a given input PTA $\mathscr{A}$ and a reference valuation π_0, the output of $IM(\mathscr{A}, \pi_0)$ is not necessarily always the same. This comes in particular from the nondeterministic choice of the π_0-incompatible inequality J to negate in the algorithm (see line 5 in Algorithm 1 page 45).

Proposition 3.12 (Non-confluence)**.** *There exist a PTA $\mathscr{A}$ and a reference valuation π_0 such that the output of $IM(\mathscr{A}, \pi_0)$ is not always the same.*

Proof. Consider the PTA depicted in Figure 3.8. This PTA contains 2 locations q_0 and q_1, one clock x, and 3 parameters p_1, p_2, p_3.

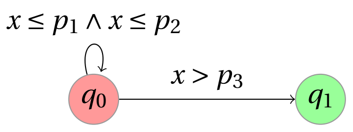


Figure 3.8: PTA showing the non-confluence of algorithm *IM*

We consider the following reference valuation π_0 of the parameters:

$$p_1 = 1 \ \wedge \ p_2 = 1 \ \wedge \ p_3 = 2$$

It is easy to see that an application of *IM* to this PTA and this valuation π_0 will output a constraint K_0 either equal to $p_1 \leq p_3$, or equal to $p_2 \leq p_3$, depending on which inequality $\neg J$ is selected in the algorithm. ■

Non-maximality. It follows from this property of non-confluence that the constraint K_0 output by *IM* is not *maximal*, i.e., there may exist $\pi \not\models K_0$ such that the traces of $\mathscr{A}[\pi_0]$ and the traces of $\mathscr{A}[\pi]$ are identical. The maximal constraint is actually not necessarily in conjunctive form in the general case: on the example of Figure 3.8, it is easy to see that the maximal constraint guaranteeing the same behavior as under π_0 is $p_1 \leq p_3 \vee p_2 \leq p_3$, which is not in conjunctive form.

This can be stated more formally in the following corollary.

Corollary 3.13 (Non-maximality)**.** *Let $\mathscr{A}$ be a PTA, and π_0 be a valuation of the parameters. Then there may exist $\pi \not\models K_0$ such that the trace sets of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$ are the same.*

Proof. From Proposition 3.12. ■

A variant of the inverse method synthesizing constraints in a non-conjunctive form is presented in Section 3.5.2. However, the constraint output by this variant is not maximal either.

LTL-equivalence. We are now interested in studying which properties on trace sets are preserved by our inverse method. Actually, because the inverse method guarantees the *equality* of trace sets, all properties on *traces* are preserved. This in particular the case of linear-time properties and, more specifically, of properties specified using the Linear Temporal Logic. This logic is an extension of the propositional logic with modalities, allowing to express properties on the execution of an infinite reactive system, such as the safety, the fairness or the liveness (see, e.g., [BK08]). Properties expressed using LTL can express, for example, the fact that atomic propositions occur *always*, *eventually*, at the *next* state, or hold for all states from the current state *until* another atomic proposition holds. Since this thesis is not centered on LTL, we will not go further into details. For a brief reminder of the syntax and the semantics of LTL properties, refer to Appendix A.1.

The LTL logic requires a set of atomic propositions, which does not appear in our formalism of trace sets. Several possibilities can be used to overcome this aspect. First, we may consider only the locations of our trace sets, and thus abstract again from the alternating sequence of locations and actions. In that case, we can either change the formalism of our PTAs and add to the tuple defining a PTA a labeling function from the locations to a given set of atomic propositions, or simply consider that our locations *are* the atomic propositions themselves. Second, we can consider that the atomic propositions are the actions. This case is very similar to the first case. The last option is to consider both locations and actions, which is equivalent to the two previous cases, but needs more tricky considerations [BK08]. We will consider in the following that the atomic propositions are the locations themselves. Our results can be extended in a straightforward manner to the other cases, because our method guarantees the equality of trace sets, and thus of any abstraction made of those trace sets.

Now, recall that, from their intrinsic definition, the traces we consider are *finite* (see Definition 2.14). As a consequence, the formulae preserved by the inverse method are only the formulae involving properties verifiable using finite traces only. Such properties correspond to the reachability and safety properties. However, the fairness or liveness properties, which check if the system goes infinitely often in a given location, do not apply to finite traces. Extending the correctness of our inverse method to infinite traces, and thus preserving full LTL properties, is the subject of future work (see Remark 3.16 below).

We first define the satisfiability of a LTL-formula by a TA.

Definition 3.14 (Satisfiability)**.** Let φ be a LTL formula verifiable using finite traces.

Let $T = q_0 \overset{a_0}{\Rightarrow} q_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{n-1}}{\Rightarrow} q_n$ be a (finite) trace. We say that T satisfies φ, denoted by $T \models \varphi$, if $q_0 q_1 \dots q_n \models \varphi$ (as defined in Appendix A.1).

Let TS be a trace set. We say that TS satisfies φ, denoted by $TS \models \varphi$ if, for all $T \in TS$, $T \models \varphi$.

Let $\mathcal{A}$ be a TA. We say that $\mathcal{A}$ satisfies φ, denoted by $\mathcal{A} \models \varphi$ if its trace set satisfies φ. ■

We can now state the preservation of the satisfiability of LTL-formulae for finite traces by the constraint synthesized by our inverse method.

Proposition 3.15 (LTL-Equivalence)**.** *Let $\mathcal{A}$ be a PTA, π_0 a valuation of the parameters, and φ a LTL formula verifiable using finite traces. Let $K_0 = IM(\mathcal{A}, \pi_0)$. Then, for all $\pi \models K_0$, $\mathcal{A}[\pi] \models \varphi$ if and only if $\mathcal{A}[\pi_0] \models \varphi$.*

Proof. From Theorem 3.8 guaranteeing the equality of trace sets. ■

Remark 3.16 (Preservation of full LTL). It would be interesting to show the preservation (or possibly the non-preservation) of the full set of LTL properties, i.e., the equivalence of sets of *infinite* traces. This could be done using theorem 3.30 of [BK08], relating finite trace and trace inclusion. However, applying directly this theorem to our framework does not hold, because this theorem supposes that the trace sets have a finite branching structure, and contain no terminal states. As a consequence, although it seems to be an interesting direction of research, proving the result would deserve further investigation. □

Non-preservation of bisimulation. We saw that our equality of trace sets preserves LTL formulae verifiable using finite traces. However, because the inverse method preserves only the trace sets and not necessarily the branching structure, the inverse method does not necessarily preserve formulae expressed using branching logics. In particular, we show below using a counter-example that the Computation Tree Logic (CTL) is not preserved. We briefly recall the CTL syntax and semantics in Appendix A.2.

Proposition 3.17 (Non-CTL-Equivalence)**.** *Let $\mathscr{A}$ be a PTA, and π_0 be a valuation of the parameters. Let $K_0 = IM(\mathscr{A}, \pi_0)$. Then there may exist a CTL-formula which is true for $\mathscr{A}[\pi_0]$, and not for $\mathscr{A}[\pi]$ for some $\pi \models K_0$.*

Proof. Consider the PTA $\mathscr{A}$ depicted in Figure 3.9.

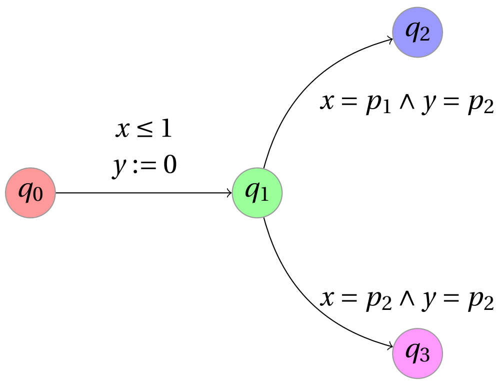


Figure 3.9: PTA showing the non-CTL-equivalence of *IM*

We consider the following reference valuation of the parameters:

$$\pi_0 : p_1 = 2 \wedge p_2 = 1$$

One can easily see that, for this example, the inverse method algorithm $IM(\mathscr{A}, \pi_0)$ outputs the following constraint K_0:

$$p_2 \leq p_1 \wedge p_1 \leq p_2 + 1$$

Now, consider the following CTL formula φ:

$$\exists \bigcirc (\exists \bigcirc q_2 \wedge \exists \bigcirc q_3)$$

This formula says that, from the initial state, there exists a next state such that, from this state, there exist both a next state labeled with q_2 *and* a next state labeled with q_3. The conjunction implies that there exists a run from q_0 such that *both* q_2 and q_3 *must* be reachable from q_1.

Recall that the CTL logic relates to the *branching* behavior of a system. As a consequence, it is not enough to check the trace set, but we need also to have a look at the semantics of the system in terms of a labeled transition system, i.e., taking into account the value of the clocks.

We will show that (1) the formula φ does not hold for $\mathscr{A}[\pi_0]$, and that (2) there exists $\pi \models K_0$ such that the formula φ holds for $\mathscr{A}[\pi]$. Actually, the duration of the stay in the initial location q_0 will impact the behavior of the system.

(1) Behavior under π_0. Three different cases are considered for π_0, depending on the duration of the stay in q_0.

1. If an execution stays in q_0 for a null duration, once in q_1 it is possible to go only to q_3, at time $t = p_2 = 1$. However, it is impossible to reach q_2, since $x = y$ but $p_1 \neq p_2$. Thus, φ does not hold.

2. If an execution stays exactly 1 unit of time in q_0, once in q_1 it is possible to go only to q_2, at time $t = 2$ (thus after 1 unit of time in q_1). In that case, $y = p_2 = 1$ and $x = p_1 = 2$. Thus, φ does not hold.

3. For any other duration in q_0, the system gets deadlocked in q_1 because neither the guard of the transition to q_2 nor the guard of the transition to q_3 will ever be satisfied. Thus, φ does not hold.

(2) Behavior under π. Let us consider the following valuation π of the parameters:

$$\pi : p_1 = 1 \wedge p_2 = 1$$

Note that we have $\pi \models K_0$. Consider the case where an execution stays in q_0 for a null duration. Then, if one stays in q_1 for exactly one unit of time, both the guard of the transition to q_2 and the guard of the transition to q_3 are satisfied. Thus, we found a run such that both q_2 and q_3 are reachable from q_1. Thus, φ holds.

Actually, more generally, it can be shown that the formula φ holds if and only if $p_1 = p_2$. ■

As a consequence, one can show that the algorithm *IM* does not involve a (time-abstract) bisimulation, i.e., there is no bisimulation (in the general case) between $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$, for all $\pi \models IM(\mathscr{A}, \pi_0)$. We briefly recall the notion of time-abstract bisimulation relation in Appendix A.3.

Proposition 3.18 (Non-bisimulation)**.** *There exist a PTA $\mathscr{A}$, a valuation π_0 of the parameters, and a valuation $\pi \models IM(\mathscr{A}, \pi_0)$ such that there is no time-abstract bisimulation between $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$.*

Proof. By Proposition 3.17 and equivalence between CTL-equivalence and bisimulation [BCG88]. ■

Remark 3.19. It may be useful to mention that our inverse method does also not preserve properties expressed using the TCTL logic. Recall that TCTL [ACD93] is a timed extension of CTL to the timed case, allowing to express properties specifying both the branching behavior and interval of time within which events may occur. TCTL properties are not preserved by our inverse method for two reasons. First, TCTL is a timed extension of CTL, based on the branching structure of the system. Our inverse method does not preserve CTL, and will not preserve TCTL for the same reasons. Second, TCTL properties are not time-abstract properties, because they express facts involving time, such as "an event must occur within a 2 seconds". Our inverse method is based on time-abstract traces, and therefore cannot guarantee any *timed* behavior, but only the preservation of the ordering between events. □

Commutation of the Instantiation. We finally show a result stating the commutation of the instantiation of some parameters with the application of the inverse method. In other words, we show that the application of the inverse method to a system in which some of the parameters are instantiated is equivalent to applying the inverse method to the fully parametric model, and then instantiating some of the parameters in the resulting constraint.

Let us first introduce some notation. Let $\sigma : P \rightharpoonup \mathbb{Q}_{\geq 0}$ be a partial function assigning a rational value to some of the parameters P (we assume that $\rightharpoonup$ is the operator for partial functions). Then, given a PTA $\mathscr{A}$, we denote by $\mathscr{A}_{/\sigma}$ the PTA obtained from $\mathscr{A}$ by replacing, for each parameter p_i for which σ is defined, any occurrence of p_i within guards and invariants of $\mathscr{A}$ by $\sigma(p_i)$. Similarly, given a constraint K on the parameters, we denote by $K_{/\sigma}$ the constraint obtained from K by replacing, for each parameter p_i for which σ is defined, any occurrence of p_i by $\sigma(p_i)$. Observe that $K_{/\sigma} = K \wedge \bigwedge_{p_i \in D_\sigma} p_i = \sigma(p_i)$, where D_σ denotes the domain of σ, i.e., the set of parameters for which σ is defined.

We can now state more formally this result below.

Proposition 3.20 (Commutation)**.** *Let $\mathscr{A}$ be a PTA, and π_0 be a valuation of the parameters. Let $\sigma : P \rightharpoonup \mathbb{Q}_{\geq 0}$ be a partial valuation of the parameters P. Then, we have:*

$$IM(\mathscr{A}_{/\sigma}, \pi_0) = (IM(\mathscr{A}, \pi_0))_{/\sigma}.$$

The proof of this proposition is rather straightforward, and is based on the equivalence of the application of the inverse method to the partially instantiated PTA, and of the partial instantiation of the constraint resulting from the inverse method applied to the non-instantiated PTA. More details can be found in [Sou10c].

This result has two implication. First, when one is interested into optimizing only *some* of the timing bounds, it does not bring anything to apply the inverse method to the fully parameterized system, and then instantiating all the parameters but these timing bounds. One can directly apply it to the "semi-instantiated" system where only the timing bounds we are interested to optimize are parametric. Actually, those two different techniques are also almost equivalent in practice, when using the implementation IMITATOR II (see Chapter 4).

Second, it may also be interesting to reason in a fully parametric way. Consider the case of the Root Contention Protocol [HRSV02, CS01], which will be studied in Section 4.4, and in Chapter 6. The IEEE reference valuation of the parameters prevents the use of the probabilistic model checker PRISM for computing probabilities because some of the instantiated parameters were too large (e.g., $s\_min = 1590$) to be handled by PRISM. Instead, a fully-parametrized analysis allowed the authors to rescale all the parameters to as low as desired (around 1) so that PRISM would be able to handle the analysis. In our case, we see that the application of the inverse method to a semi-instantiated model (depicted as tile 1 in Figure 6.11 page 166) allows us to rescale s_min to around 850. But the application of the inverse method to the fully parametrized model allows us to rescale s_min as low as one wants, e.g., to 7 (see Section 4.4, as well as Table 6.1 page 159), allowing a much faster computation by PRISM.

3.3.4 Discussion

The first advantage of the inverse method is that it gives a criterion of *robustness* by ensuring the correctness of the system for other values for the parameters around the reference valuation. This is of interest when implementing a system: indeed, the exact model with (for example) integer values for timing delays that has been formally verified will necessarily be implemented using values which will not be exactly the ones that have been verified.

Moreover, it allows the system designer to *optimize* the value of some parameters without changing the overall functional behavior of the system. By instantiating within the constraint output by the algorithm all but some parameters, one can get lower (or upper) bounds on their possible values, and thus optimize them without changing the time-abstract behavior of the system. This has numerous applications, especially in hardware verification (see Chapter 4).

A further advantage of the method is to allow the *rescaling* of constants. Indeed, it is possible that verifying a timed concurrent system using an external model checker is sensitive to the size of the constants. As a consequence, for systems with large constants that one can hardly verify using external tools, it is interesting to run the inverse method implemented to get a constraint. Then, one can infer from this constraint much smaller values of the parameters having the same time-abstract behavior as under the original (large) valuation. The verification of such a *rescaled* system can then lead to a high decrease of the verification time and the state space.

Also observe that the inverse method does not depend on a property one wants to check; actually, the algorithm does not take into account the fact that the trace set under the reference valuation is good or bad. Actually, although the final constraint K_0 output by the inverse method induces a behavioral property of the system related to traces, only states (and not traces) are manipulated by the algorithm.

Although most examples detailed in this thesis are relevant to the synthesis of good parameters, the method can also be applied to synthesize parameters with a “bad” behavior. In particular, it can be interesting, once one has discovered a parameter valuation leading to a bad behavior, to find other such parameter valuations; an application can be to find the *smallest* parameter valuation leading to a given bad behavior.

The main shortcoming of the inverse method is that the constraint output by the algorithm is not maximal, i.e., there may exist other parameter valuations outside K_0 with the same trace set as under the reference valuation (see Proposition 3.13). Moreover, recall that the good parameters problem we are interested to solve in this thesis relates to the synthesis of parameter valuations corresponding to *any* good behavior, not to a single one. There may actually exist *different* good behaviors from the one corresponding to the reference valuation. We will address those issues in Chapter 5.

Rewriting the Algorithm. The inverse method algorithm has been presented as such in Algorithm 1 both for historical and correctness reasons. Indeed, although several computations are redundant, it was rather easy to prove the correctness of *IM* using that version. However, the inverse method can be rewritten in a simpler manner, saving two variables (viz., K and i). This version, which is close to the version implemented, is sketched in Section 4.1.2.

3.4 Application to the Flip-flop Example

Let us now apply the inverse method to the flip-flop example of Section 3.1.1. Applying the inverse method algorithm to this model, the following constraint K_0 is computed after 9 iterations:

$$\begin{array}{rrcr} & \delta_3^+ + \delta_4^+ \geq T_{Hold} & \wedge & \delta_1^- > 0 \\ \wedge & T_{Hold} \geq \delta_3^- + \delta_4^- & \wedge & T_{Hold} > \delta_3^+ \\ \wedge & T_{HI} > \delta_3^+ + \delta_4^+ & \wedge & T_{Setup} > \delta_1^+ \end{array}$$

It can be checked (using, e.g., a parametric reachability analysis) that the trace sets of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$ are the same, for any $\pi \models K_0$. This trace set corresponds to the one depicted in Figure 3.1 page 43.

Note that this constraint K_0 guarantees a good behavior of the system, because the behavior of $\mathscr{A}[\pi_0]$ is a good behavior (see Section 3.1.1). However, there may exist *other* good behaviors for this system. Finding the maximal set of parameter valuation corresponding to good behaviors will be the purpose of Chapter 5.

Comparison. In [CC07], a constraint, say Z, is synthesized in order to prevent bad system behaviors. The bad state is defined as the case where $CK^{\downarrow}$ occurs before $Q^{\uparrow}$. This constraint Z is the following:

$$\begin{array}{rlcl} & T_{CK \rightarrow Q} \leq \delta_2^+ + \delta_3^+ + \delta_4^+ & \wedge & T_{Setup} > \delta_1^+ + \delta_2^+ - \delta_2^- \\ \wedge & T_{Hold} > \delta_2^+ + \delta_3^+ & \wedge & T_{HI} > \delta_2^+ + \delta_3^+ + \delta_4^+ \\ \wedge & T_{HI} > T_{Hold} & \wedge & T_{LO} > T_{Setup} \\ \wedge & \delta_1^- > \delta_2^+ & & \end{array}$$

One can see that Z and K_0 are incomparable, i.e., $Z \not\subseteq K_0$ and $K_0 \not\subseteq Z$. The method introduced in Chapter 5 will allow us to synthesize a constraint strictly weaker (i.e., containing a strictly larger set of parameter valuations) than Z for 2 parameter dimensions (see Section 5.4.3).

3.5 Variants of the Inverse Method

The standard inverse method algorithm *IM* presented above guarantees the equality of trace sets. This guarantee can sometimes be considered as too strong for what the designer is interested in. In particular, this equality of trace sets is too strong with respect to our good parameters problem, as defined in Section 2.3. Indeed, the good parameters problem focuses on the synthesis of all good behaviors, not only one specific good behavior.

We present in this section two variants of the inverse method, not anymore guaranteeing the equality of trace sets, but still featuring interesting properties when one is interested in synthesizing good values for the parameters.

3.5.1 Variant with State Inclusion in the Fixpoint

As shown in line 8 in Algorithm 1, the standard inverse method algorithm stops when $Post_{\mathscr{A}(K)}(S) \sqsubseteq S$, i.e., when each of the new states encountered at step i

is *equal* to a previously encountered state (see Definition 2.30). A variant of the algorithm consists in considering an inclusion of the states, i.e., each of the new states encountered at step i is *included* in a previously encountered state (recall that state inclusion means equality of locations and inclusion of constraints, see Definition 2.29).

We give in Algorithm 2 the lines replacing lines 8 and 9 of Algorithm 1, thus obtaining variant $IM^{\subseteq}$.

Algorithm 2: Variant $IM^{\subseteq}(\mathscr{A},\pi)$ of the inverse method

```
1 if $\forall s \in Post_{\mathscr{A}(K)}(S), \exists s' \in S : s \subseteq s'$ then
2     return $\bigcap_{(q,C)\in S}(\exists X : C)$
```

Note that the computation of the constraint returned by the algorithm is not modified itself; only the termination condition is.

Termination for this variant $IM^{\subseteq}$ may happen earlier than using the standard algorithm IM, due to this weaker condition of fixpoint. The implementation also uses less memory, because one can merge states as soon as one is included into another one. As a consequence, this variant $IM^{\subseteq}$ may terminate in some cases where the standard algorithm IM does not terminate. This better termination condition is a major advantage of this variant. The constraint output is also weaker than the constraint output by the standard algorithm IM.

Although the equality of trace sets stated by Theorem 3.8 for IM does obviously not hold any longer for $IM^{\subseteq}$, this variant still has interesting properties. First, we will show that the trace sets of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$ are equal up to length n, where n is the number of iterations of $IM^{\subseteq}(\mathscr{A},\pi_0)$. In other words, the set of prefixes of length n of the traces of $\mathscr{A}[\pi_0]$ is equal to the set of prefixes of length n of the traces of $\mathscr{A}[\pi]$. We formalize this property in Proposition 3.26. Second, we will show that the non-reachability of a location is preserved, i.e., a location which does not belong to the trace set of $\mathscr{A}[\pi_0]$ does also not belong to the trace set of $\mathscr{A}[\pi]$, for all $\pi \models IM^{\subseteq}(\mathscr{A},\pi_0)$. We formalize this property in Proposition 3.27.

Following a reasoning similar to the proof of Theorem 3.8 for the algorithm IM, we first need to prove several propositions. In the following, we consider given a PTA $\mathscr{A}$, and a reference valuation π_0. We denote by n the number of iterations of $IM^{\subseteq}(\mathscr{A},\pi_0)$, and by K the constraint at the end of the algorithm.

We first show that $\pi_0 \models K_0$.

Lemma 3.21. *Let $\mathscr{A}$ be a PTA, and π_0 be a valuation of the parameters. Let $K_0 = IM^{\subseteq}(\mathscr{A},\pi_0)$. Then, we have $\pi_0 \models K_0$.*

Proof. When Algorithm $IM^{\subseteq}$ terminates, the set S is π_0-compatible (i.e., $\pi_0 \models (\exists X : C)$, for all $(q, C) \in S$). Thus, the intersection K_0 of the constraints associated with the states of S, i.e., $\bigcap_{(q,C)\in S}(\exists X : C)$, is satisfied by π_0. ■

We then prove the following lemma, used to prove Proposition 3.26.

Lemma 3.22. *Let $\mathscr{A}$ be a PTA, and π_0 be a valuation of the parameters. Let $K_0 = IM^{\subseteq}(\mathscr{A}, \pi_0)$. Then, we have $K_0 \subseteq K$, where K corresponds to the negation of all inequalities at the end of the algorithm.*

Proof. From Lemma 2.35, for all states $(q, C) \in S$, we have $(\exists X : C) \subseteq K$, since $S = Post^n_{\mathscr{A}(K)}(\{s_0\})$. As $K_0 = \bigcap_{(q,C)\in S}(\exists X : C)$, then $K_0 \subseteq K$. ■

We now state that, for all $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$ of length smaller or equal to n, we can find an equivalent concrete run of $\mathscr{A}[\pi]$.

Lemma 3.23. *For all π such that $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$ of length smaller or equal to n reaching (q, C), there exists a clock valuation w such that $<w, \pi> \models C$.*

Proof. For each symbolic run of $\mathscr{A}(K)$ of length smaller or equal to n reaching (q, C), we have $(q, C) \in S$ since $S = Post^n_{\mathscr{A}(K)}(\{s_0\})$. Moreover, we have $K_0 = \bigcap_{(q,C)\in S}(\exists X : C)$. Thus, for all $\pi \models K_0$, for all $(q, C) \in S$, we have $\pi \models (\exists X : C)$. Hence, there exists a clock valuation w such that $<w, \pi> \models C$. ■

Proposition 3.24. *For all $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$ of length smaller or equal to n, there exists an equivalent concrete run of $\mathscr{A}[\pi]$.*

Proof. From Lemma 3.23 and Lemma 3.4. ■

Conversely, we now state that, for all $\pi \models K_0$, for each concrete run of $\mathscr{A}[\pi]$ of length smaller or equal to n, we can find an equivalent symbolic run of $\mathscr{A}(K)$.

Proposition 3.25. *For all $\pi \models K_0$, for each concrete runs of $\mathscr{A}[\pi]$ of length smaller or equal to n, there exists an equivalent symbolic run of $\mathscr{A}(K)$.*

Proof. The proof of Proposition 3.18 in [HRSV02] can be adapted in a straightforward manner to show that, for all $\pi \models K$, for each concrete run of $\mathscr{A}[\pi]$ of length smaller or equal to n, there exists an equivalent symbolic run of $\mathscr{A}(K)$. The result follows from the fact that $\pi \models K_0$ implies $\pi \models K$ (by Lemma 3.22). ■

We now show Proposition 3.26, stating that the set of prefixes of length n of the traces of $\mathscr{A}[\pi_0]$ is equal to the set of prefixes of length n of the traces of $\mathscr{A}[\pi]$, where n is the number of iterations of $IM^{\subseteq}$ (i.e., the value of i at the end of the algorithm).

Proposition 3.26 (Prefix traces)**.** *Let $\mathcal{A}$ be a PTA, and π_0 a valuation of the parameters. Suppose that $IM^{\subseteq}(\mathcal{A}, \pi_0)$ terminates with output K_0 after n iterations of the outer* **do** *loop. Then, we have:*

1. *$\pi_0 \models K_0$,*

2. *for all $\pi \models K_0$, for each trace T_0 of $\mathcal{A}[\pi_0]$, there exists a trace T of $\mathcal{A}[\pi]$ such that the prefix of length n of T_0 and the prefix of length n of T are equal, and*

3. *for all $\pi \models K_0$, for each trace T of $\mathcal{A}[\pi]$, there exists a trace T_0 of $\mathcal{A}[\pi_0]$ such that the prefix of length n of T_0 and the prefix of length n of T are equal.*

Proof. From Lemma 3.21, Proposition 3.24 and Proposition 3.25. ■

When considering the absence of bad behavior, i.e., the non-reachability of a given location, the above proposition only shows that, if a location is not reachable in $\mathcal{A}[\pi_0]$, it is also not reachable in $\mathcal{A}[\pi]$, for all $\pi \models IM^{\subseteq}(\mathcal{A}, \pi_0)$, *within traces up to length n*. Although this may be of interest for bounded verification, it is usually more interesting to guarantee that a given location is *never* reachable, i.e., does not belong to any trace of unbounded length. We now show that, if a given location is not reachable in $\mathcal{A}[\pi_0]$, it will also not be reachable in $\mathcal{A}[\pi]$, for all $\pi \models IM^{\subseteq}(\mathcal{A}, \pi_0)$. Besides the weaker constraint and the better termination, this is certainly the most interesting property of this variant $IM^{\subseteq}$.

Proposition 3.27 (Preservation of non-reachability)**.** *Let $\mathcal{A}$ be a PTA, π_0 a valuation of the parameters, and q a location of $\mathcal{A}$. Suppose that $IM^{\subseteq}(\mathcal{A}, \pi_0)$ terminates with output K_0. If q does not belong to the trace set of $\mathcal{A}[\pi_0]$, then q does not belong to the trace set of $\mathcal{A}[\pi]$, for all $\pi \models K_0$.*

Proof. Let $\mathcal{A}$ be a PTA, and π_0 be a valuation of the parameters. Suppose that $IM^{\subseteq}(\mathcal{A}, \pi_0)$ terminates with output K_0 after n iterations. Let $\pi \models K_0$. Let q be a location of $\mathcal{A}$ such that q does not belong to the trace set of $\mathcal{A}[\pi_0]$. By Proposition 3.26, q does not belong to any trace of $\mathcal{A}[\pi]$ of length lower or equal to n.

Moreover, from the fixpoint of the algorithm $IM^{\subseteq}$, all states reached after n iterations are *included* into states computed previously. Recall from Definition 2.29 that the state inclusion involves equality of locations and inclusion of constraints. From the semantics of PTAs, the states reachable from those new states will themselves be included into states computed previously. As a consequence, no location not reachable previously can be reached after n iterations, which proves the proposition. ■

Non-maximality. It can be shown (e.g., using the PTA depicted in Figure 3.8 page 54 and used to prove the non-confluence of the standard version *IM*) that this variant $IM^{\subseteq}$ is also non-confluent. As a consequence, the constraint output by $IM^{\subseteq}$ is also non-maximal. Still, the constraint output by $IM^{\subseteq}$ is weaker than the one output by *IM* for the same input (see Proposition 3.28 below).

Advantages. Although the equality of trace sets is not guaranteed anymore, this variant $IM^{\subseteq}$ is interesting when one is interested in safety properties. Indeed, if a given "bad" location is not reached under a reference valuation, it will also not be reached under any valuation satisfying the constraint output by this variant. Moreover, termination is ensured more often than for *IM*.

Proposition 3.28. *Let $\mathscr{A}$ be a PTA, and π_0 a valuation of the parameters. If $IM(\mathscr{A}, \pi_0)$ terminates, then $IM^{\subseteq}(\mathscr{A}, \pi_0)$ also terminates.*

Proof. Based on the fact that the two algorithms are equal, except the fixpoint condition, which is weaker for $IM^{\subseteq}$. ■

Note that the reciprocal statement does not hold: there are actually examples of PTAs for which the application of $IM^{\subseteq}$ terminates although *IM* does not terminate.

Example 3.29. Consider again the PTA depicted in Figure 3.7 page 54. As said in Example 3.11, the application of *IM* to this PTA does not terminate. However, the application of $IM^{\subseteq}$ to this PTA and any reference valuation of the parameters terminates (and outputs the constraint `true`). □

Also note that, even in the cases where termination occurs for both *IM* and $IM^{\subseteq}$, termination of $IM^{\subseteq}$ may occur earlier (in term of number of iterations) than *IM* because of the weaker fixpoint condition.

Finally, we state in the following proposition that the constraint output by $IM^{\subseteq}$ is weaker (i.e., corresponds to a larger set of parameter valuations) than the one output by *IM*.

Proposition 3.30. *Let $\mathscr{A}$ be a PTA, and π_0 a valuation of the parameters. Then, we have:*

$$IM(\mathscr{A}, \pi_0) \subseteq IM^{\subseteq}(\mathscr{A}, \pi_0).$$

Proof. Suppose $IM^{\subseteq}(\mathscr{A}, \pi_0)$ terminates after n iterations. Up to the nth iteration, *IM* behaves exactly like $IM^{\subseteq}$ (because both algorithms are identical, except the fixpoint condition). After the nth iteration, K may be restricted in *IM* with the addition of inequalities. Moreover, new states may be computed, but their associated constraint on the parameters will be more restrictive than (i.e., included into) the constraints associated with the states of S at the nth iteration (by Lemma 2.34). Thus, the intersection of K with the constraints on the parameters associated with the reachable states is included into this same intersection at the nth iteration. ■

This variant has been implemented (see Chapter 4) and applied to several case studies.

3.5.2 Variant with Union of the Constraints

As shown in line 9 in Algorithm 1, the standard inverse method algorithm returns the intersection of the constraints associated with *all* the reachable states, when the fixpoint is reached. As a consequence, all the states reachable under π_0 will be reachable, for any $\pi \models K_0$.

Let us now consider what would happen if one replaces the intersection of all the constraints with the *union* of all the constraints. Actually, we do not consider the union of the constraints associated with all the reachable states, but the union of the constraints associated with each of the *last* state of a symbolic run. This notion of last state is easy to understand for finite symbolic runs. However, when considering infinite (and necessarily cyclic) symbolic runs, the last state refers to the second occurrence of a same state within a symbolic run, i.e., to the first time that a state is equal to a previous state of the same symbolic run. Such symbolic runs are necessarily cyclic because, if there are any infinite runs with diverging states, then the algorithm does not terminate.

We give this variant $IM^{\cup}$ in Algorithm 3. This algorithm is identical to the standard inverse method IM (see Algorithm 1) except in two points:

1. Lines 8 to 10 allow the computation of the last states of each run: either states being the last of a finite run (i.e., when $Post_{\mathscr{A}(K)}(\{s\})$ is empty), or states being equal to states computed previously (i.e., when $s \in S$).

2. Line 12 returns the union of the constraints on the parameters associated with the last states of the runs, instead of the intersection of the constraints associated with all the states computed.

Note that the lines mentioned at item 1 are actually modified in this algorithm only in order to compute the different return of item 2, and do not interfere with the rest of the algorithm.

Correctness. Although it is clear that the equality of trace sets is no longer guaranteed for $\pi \models IM^{\cup}(\mathscr{A}, \pi_0)$, some properties are still preserved by this variant. By performing the union of the last states of each trace, we have the guarantee that, for all $\pi \models K_0$, the trace set of $\mathscr{A}[\pi]$ is a subset of the trace set

Algorithm 3: Variant $IM^{\cup}(\mathscr{A}, \pi)$ of the inverse method

```
input  : A PTA $\mathscr{A}$ of initial state $s_0 = (q_0, C_0)$
input  : Valuation $\pi$ of the parameters
output: Constraint $K_0$ on the parameters

1  $i \leftarrow 0$;  $K \leftarrow$ true;  $S \leftarrow \{s_0\}$;  $S_{last} \leftarrow \{\}$
2  while true do
3      while there are $\pi$-incompatible states in $S$ do
4          Select a $\pi$-incompatible state $(q, C)$ of $S$ (i.e., s.t. $\pi \not\models C$) ;
5          Select a $\pi$-incompatible $J$ in $(\exists X : C)$ (i.e., s.t. $\pi \not\models J$) ;
6          $K \leftarrow K \wedge \neg J$ ;
7          $S \leftarrow \bigcup_{j=0}^{i} Post^{j}_{\mathscr{A}(K)}(\{s_0\})$ ;
8      foreach $s \in Post_{\mathscr{A}(K)}(S)$ do
9          if $Post_{\mathscr{A}(K)}(\{s\}) = \emptyset$ or $s \in S$ then
10             $S_{last} \leftarrow S_{last} \cup \{s\}$
11     if $Post_{\mathscr{A}(K)}(S) \sqsubseteq S$ then
12         return $\bigcup_{(q,C) \in S_{last}} (\exists X : C)$
13     $i \leftarrow i + 1$ ;
14     $S \leftarrow S \cup Post_{\mathscr{A}(K)}(S)$ ;                // $S = \bigcup_{j=0}^{i} Post^{j}_{\mathscr{A}(K)}(\{s_0\})$
```

of $\mathcal{A}[\pi_0]$. In other words, for all $\pi \models K_0$, for each trace of the trace set of $\mathcal{A}[\pi]$, there exists an identical trace in the trace set of $\mathcal{A}[\pi_0]$.

This gives in particular a criterion of *safety* by guaranteeing that a "bad" location which is not reachable under π_0 is also not reachable under π.

In the following, we consider given a PTA $\mathcal{A}$, and a reference valuation π_0. We denote by K the constraint at the end of the algorithm.

We first prove that $\pi_0 \models K_0$.

Lemma 3.31. *We have $\pi_0 \models K_0$.*

Proof. By construction, the states in S_{last} are π_0-compatible. Moreover, the constraint K is made by construction of negations of π_0-incompatible inequalities, and is thus also π_0-compatible. As a consequence, K_0 is π_0-compatible. ■

We now prove that $K_0 \subseteq K$.

Lemma 3.32. *Let $\mathcal{A}$ be a PTA, and π_0 be a valuation of the parameters. Let $K_0 = IM^{\cup}(\mathcal{A}, \pi_0)$. Then, we have $K_0 \subseteq K$.*

Proof. From Lemma 2.35, for each state $(q, C) \in S$, we have $(\exists X : C) \subseteq K$, since $S = Post^*_{\mathcal{A}(K)}(\{s_0\})$. As $K_0 = \bigcup_{(q,C)\in S_{last}}(\exists X : C)$, with $S_{last} \subseteq S$, then $K_0 \subseteq K$. ■

Now, recall that we stated both for the standard version *IM* (Proposition 3.5) and for the variant $IM^{\subseteq}$ (Proposition 3.24) that, for all $\pi \models K_0$, for each symbolic run of $\mathcal{A}(K)$, we can find an equivalent concrete run of $\mathcal{A}[\pi]$. This does not hold here, because we do not necessarily have that $\pi \models s$ for all $s \in S$ (where S is the set of reachable states at the end of $IM^{\cup}$). However, we can show this for π_0, because all states are π_0-compatible at the end of $IM^{\cup}$.

Lemma 3.33. *For each symbolic run of $\mathcal{A}(K)$ reaching (q, C), there exists a clock valuation w such that $<w, \pi_0> \models C$.*

Proof. For each symbolic run of $\mathcal{A}(K)$ reaching (q, C), we have $(q, C) \in S$ since $S = Post^*_{\mathcal{A}(K)}(\{s_0\})$. By construction, all states in S are π_0-compatible, i.e., $\pi_0 \models (\exists X : C)$. Hence, there exists a clock valuation w such that $<w, \pi_0> \models C$. ■

This implies that, for each symbolic run of $\mathcal{A}(K)$, there exists an equivalent concrete run of π_0.

Proposition 3.34. *For each symbolic run of $\mathcal{A}(K)$, there exists an equivalent concrete run of $\mathcal{A}[\pi_0]$.*

Proof. From Lemma 3.33 and Lemma 3.4. ■

Although we could not show that, for all $\pi \models K_0$, for each symbolic run of $\mathscr{A}(K)$, we can find an equivalent concrete run of $\mathscr{A}[\pi]$, we can nevertheless show the converse: for all $\pi \models K_0$, for each concrete run of $\mathscr{A}[\pi]$, we can find an equivalent symbolic run of $\mathscr{A}(K)$.

Proposition 3.35. *For all $\pi \models K_0$, for each concrete run of $\mathscr{A}[\pi]$, there exists an equivalent symbolic run of $\mathscr{A}(K)$.*

Proof. The proof of Proposition 3.18 in [HRSV02] can be adapted in a straightforward manner to show that, for all $\pi \models K$, for each concrete run of $\mathscr{A}[\pi]$, there exists an equivalent symbolic run of $\mathscr{A}(K)$. The result follows from the fact that $\pi \models K_0$ implies $\pi \models K$ (by Lemma 3.32). ■

Since $\pi_0 \models K_0$, we can state the equality of trace sets between $\mathscr{A}(K)$ and $\mathscr{A}[\pi_0]$.

Proposition 3.36. *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM^{\cup}(\mathscr{A}, \pi_0)$. Then the trace sets of $\mathscr{A}(K)$ and $\mathscr{A}[\pi_0]$ are equal.*

Proof. From Lemma 3.31, we have $\pi_0 \models K_0$. The result then comes from Proposition 3.34 and Proposition 3.35. ■

This brings us to the proof of the correctness of this variant.

Theorem 3.37 (Correctness)**.** *Let $\mathscr{A}$ be a PTA and π_0 a valuation of the parameters. Let $K_0 = IM^{\cup}(\mathscr{A}, \pi_0)$. Then, for all $\pi \models K_0$, every trace of $\mathscr{A}[\pi]$ is equal to a trace of $\mathscr{A}[\pi_0]$.*

Proof. Let $\pi \models K_0$. Let r be a run of $\mathscr{A}[\pi]$. By Proposition 3.35, there exists an equivalent symbolic run r' of $\mathscr{A}(K)$. Thus the traces associated with r and r' respectively are equal. By Proposition 3.36, there exists a trace which is equal in $\mathscr{A}[\pi_0]$. ■

Now, let us show that no location unreachable under π_0 can be reached under any $\pi \models K_0$. More formally, we show in the following that, if a given location is not reachable in $\mathscr{A}[\pi_0]$, it will also not be reachable in $\mathscr{A}[\pi]$, for all $\pi \models K_0$. This proposition is actually similar to Proposition 3.27 in the framework of the variant $IM^{\subseteq}$.

Proposition 3.38 (Preservation of non-reachability)**.** *Let $\mathscr{A}$ be a PTA, π_0 a valuation of the parameters, and q a location of $\mathscr{A}$. Suppose that $IM^{\cup}(\mathscr{A}, \pi_0)$ terminates with output K_0. If q does not belong to the trace set of $\mathscr{A}[\pi_0]$, then q does not belong to the trace set of $\mathscr{A}[\pi]$, for all $\pi \models K_0$.*

Proof. By contraposition on Theorem 3.37. ■

Let us also note that the termination is the same as for *IM*.

Proposition 3.39 (Termination)**.** *Let $\mathscr{A}$ be a PTA, and π_0 a valuation of the parameters. Then, $IM^{\cup}(\mathscr{A},\pi_0)$ terminates if and only if $IM(\mathscr{A},\pi_0)$ terminates.*

Proof. Based on the fact that the fixpoint conditions of the two algorithms are the same, and the variables involved in the fixpoint (viz., S and K) are computed in the same way. ■

Non-maximality. One may wonder whether the fact that the constraint output is under a disjunctive form implies the *maximality* of the constraint. In other words, does the constraint K_0 output by $IM^{\cup}$ correspond to the maximal set of parameters valuations satisfying Theorem 3.37? The answer is no. It can be shown (e.g., using the PTA depicted in Figure 3.8 page 54 and used to prove the non-confluence of the standard version *IM*) that this variant $IM^{\cup}$ is also non-confluent. As a consequence, the constraint output by $IM^{\cup}$ is also non-maximal. Still, the constraint output by $IM^{\cup}$ is weaker than the one output by *IM* for the same input (see Proposition 3.40 below).

Advantages. Similarly to the variant $IM^{\subseteq}$, although the equality of trace sets is not guaranteed anymore, this variant $IM^{\cup}$ is interesting when one is interested in safety properties. Indeed, if a given "bad" location is not reached under a reference valuation, it will also not be reached under any valuation satisfying the constraint output by this variant.

Beside the fact that the non-reachability is preserved, the other main advantage of this variant is that the constraint corresponds to a larger set of valuations of the parameters than in the standard algorithm *IM*, because we perform the union of some states, instead of the intersection of all the reachable states. We formalize this property below.

Proposition 3.40. *Let $\mathscr{A}$ be a PTA, and π_0 a valuation of the parameters. Then:*

$$IM(\mathscr{A},\pi_0) \subseteq IM^{\cup}(\mathscr{A},\pi_0).$$

Proof. First recall that the two algorithms have the same fixpoint condition. Then, the proof is based on the fact that *IM* returns the *intersection* of *all* the states in S, whereas the variant $IM^{\cup}$ returns the *union* of *some* of the states in S. ■

However, the constraints output by the two variants $IM^{\subseteq}$ and $IM^{\cup}$ for the same input are incomparable in the general case.

Also recall that the termination of $IM^{\subseteq}$ occurs exactly in the same cases as the termination of *IM* (see Proposition 3.39).

Observe that, in contrast to the standard algorithm *IM* and to the first variant $IM^{\subseteq}$, the constraint output by this variant $IM^{\cup}$ is not under a convex form.

Combination of Variants. One can actually combine both variants $IM^{\subseteq}$ and $IM^{\cup}$, by considering the fixpoint condition of $IM^{\subseteq}$ and the return under a disjunctive form of $IM^{\cup}$. Let us name $IM^{\subseteq}_{\cup}$ this combination. This variant has the properties of both variants $IM^{\subseteq}$ and $IM^{\cup}$. In short, the constraint synthesized is weaker than the one synthesized by either *IM*, $IM^{\subseteq}$ or $IM^{\cup}$, the algorithm terminates more often that the three other versions of the algorithm, non-reachability of a location is preserved, and the constraint synthesized is still not necessarily maximal. We will not formally investigate further this variant, but we will consider it in the discussion below.

3.5.3 Discussion on the Variants

We discuss here the advantages of each variant on an example of PTA, and compare the constraints synthesized by each variant.

An Example of PTA. Let us consider the PTA $\mathscr{A}_{var}$ depicted in Figure 3.10 and containing in particular 2 parameters.

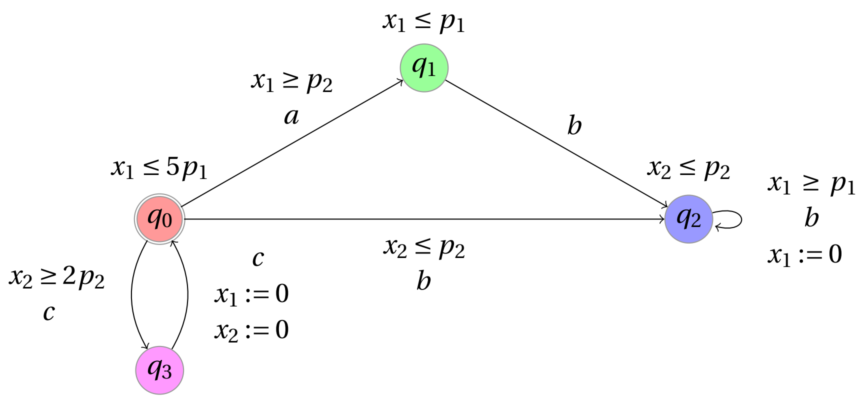


Figure 3.10: A PTA $\mathscr{A}_{var}$ for comparing the variants of *IM*

Consider the following reference valuation π_0 of the parameters:

$$p_1 = 1 \ \wedge \ p_2 = 2$$

Reference Behavior. The trace set of $\mathscr{A}_{var}$ under the reference valuation (i.e., $\mathscr{A}_{var}[\pi_0]$) is given in Figure 3.11.

Note in particular that location q_1 is not reachable under π_0.

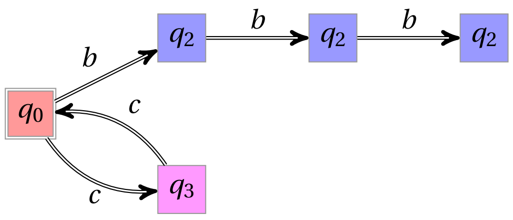


Figure 3.11: Trace set of $\mathscr{A}_{var}$ under π_0

Classical Inverse Method. Let us first apply the standard version *IM* of our inverse method to $\mathscr{A}_{var}$ and π_0. The following constraint K is synthesized:

$$p_2 \geq 2 * p_1 \ \wedge \ 3 * p_1 > p_2 \ \wedge \ 5 * p_1 \geq 2 * p_2$$

By Theorem 3.8, the trace set is exactly the same as under π_0, i.e., the one depicted in Figure 3.11.

First Variant. Now apply to $\mathscr{A}_{var}$ and π_0 the variant $IM^{\subseteq}$ of the inverse method with inclusion of constraints within the fixpoint. The following constraint $K^{\subseteq}$ is synthesized:

$$p_2 > p_1 \ \wedge \ 5 * p_1 \geq 2 * p_2$$

As stated by Proposition 3.30, it is easy to see that $K \subseteq K^{\subseteq}$.

The trace set of $\mathscr{A}_{var}[\pi]$, for any $\pi \models K^{\subseteq}$, is such that it contains the trace set of Figure 3.11, and is a sub trace set of the one given in Figure 3.12.

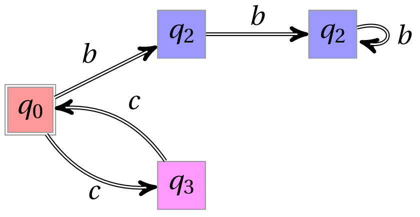


Figure 3.12: Over trace set of $\mathscr{A}_{var}[\pi]$ for $\pi \models K^{\subseteq}$

Second Variant. Now apply to $\mathscr{A}_{var}$ and π_0 the variant $IM^{\cup}$ of the inverse method returning the union of the constraints associated with the last states of traces. The following constraint $K^{\cup}$ is synthesized:

$$\begin{array}{rrcl} & p_2 \geq 2 * p_1 & \wedge & 3 * p_1 > p_2 \\ \vee & p_2 > p_1 & \wedge & 5 * p_1 \geq 2 * p_2 \end{array}$$

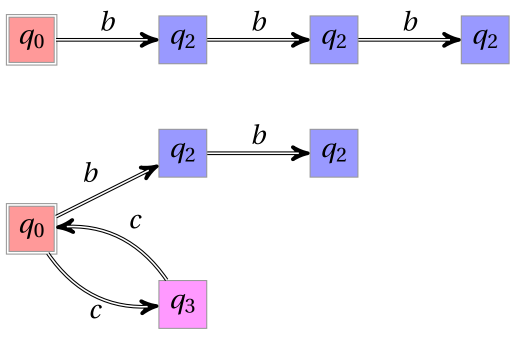


Figure 3.13: Possible trace sets of $\mathscr{A}_{var}[\pi]$ for $\pi \models K^{\cup}$

As stated by Proposition 3.40, it is easy to see that $K \subseteq K^{\cup}$.

It can be shown that the trace set of $\mathscr{A}_{var}[\pi]$, for any $\pi \models K^{\cup}$, is either the trace set given in Figure 3.11 or one of the two trace sets given in Figure 3.13. Actually, it can be shown that the trace set of $\mathscr{A}_{var}[\pi]$ is the one of Figure 3.11 for π taken in $2.5 * p_1 \geq p_2 \geq 2 * p_1$; it corresponds to the trace set of the lower part of Figure 3.13 for π taken in $2 * p_1 > p_2 > p_1$; and it corresponds to the trace set of the upper part of Figure 3.13 for π taken in $3 * p_1 > p_2 > 2.5 * p_1$.

Combined Variant. Now apply to $\mathscr{A}_{var}$ and π_0 the variant $IM^{\subseteq}_{\cup}$ of the inverse method, i.e., the variant combining $IM^{\subseteq}$ and $IM^{\subseteq}$. The following constraint $K^{\subseteq}_{\cup}$ is synthesized:

$$\begin{array}{lll} & p_2 > p_1 & \\ \vee & p_2 > p_1 & \wedge \quad 5 * p_1 \geq 2 * p_2 \end{array}$$

which is actually equivalent to $p_2 > p_1$. It is easy to see that $K \subseteq K^{\subseteq}_{\cup}$, $K^{\subseteq} \subseteq K^{\subseteq}_{\cup}$ and $K^{\cup} \subseteq K^{\subseteq}_{\cup}$.

It can be shown that the trace set of $\mathscr{A}_{var}[\pi]$, for any $\pi \models K^{\subseteq}_{\cup}$, is actually a sub trace set of the trace set given in Figure 3.12, depending on the value of π.

Comparison of the Constraints. Let us suppose that a bad behavior corresponds to the fact that a trace goes into location q_1. Under π_0, the system has a good behavior, since the trace set (see Figure 3.11) does not contain q_1. By property of the inverse method and its three variants, the constraint synthesized by any of the four versions also prevents the traces to enter q_1, by preservation of the non-reachability of a location. More generally, one can see intuitively that the parameter valuations allowing the system to enter the bad location q_1 are comprised in the domain $p_1 \geq p_2$.

Let us compare the size of the constraints synthesized. We give in Figure 3.14 the four constraints synthesized by the four versions of the inverse method. For each graphics, we depict in dark blue the parameter domain covered by the constraint, and in light red the parameter domain corresponding to a bad behavior. The "good" zone not covered by the constraint is depicted in very light gray. The point represents π_0.

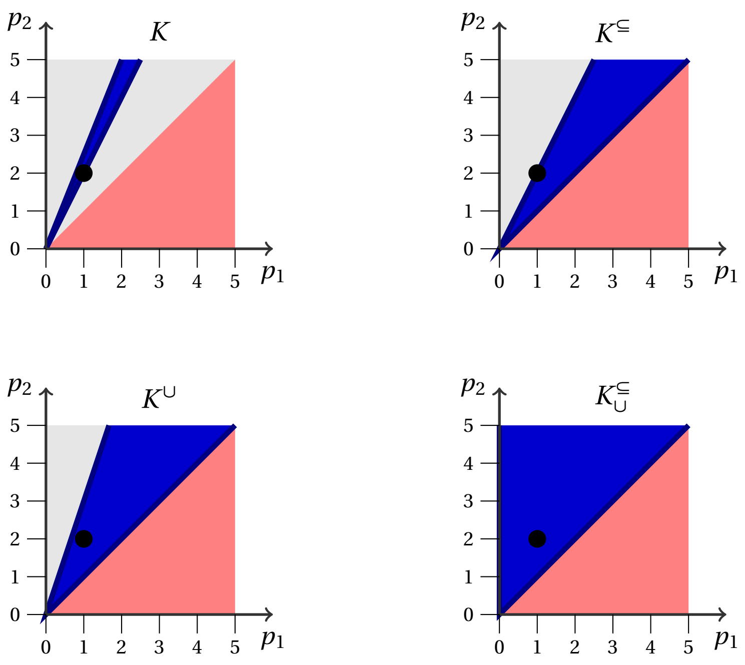


Figure 3.14: Comparison of the constraints synthesized for $\mathscr{A}_{var}$

One sees that the constraint $K^{\subseteq}_{\cup}$ is the maximal constraint guaranteeing that $\mathscr{A}_{var}$ behaves well. This is actually not the case in general. One can consider the PTA depicted in Figure 3.8 page 54 to be convinced that even $IM^{\subseteq}_{\cup}$ will not output the maximal constraint. Developing techniques resulting in the synthesis of the maximal constraint will be the purpose of Chapter 5.

3.6 Related Work

History of the Inverse Method. The inverse method has been initially proposed in the framework of "time separation of events" [EF08]. Although the framework of timed automata is more expressive and complex than the one of timing constraint graphs, and although the algorithm presented in Section 3.2 is different from the one given in [EF08], it is nevertheless certain that the underlying idea of our inverse method takes its origin in [EF08].

The "direct problem" in the framework of time separation of events can be stated as follows: "Given a system made of several connected components, each one entailing a local delay known with uncertainty, what is the maximum time for traversing the global system?" This problem is useful, e.g., in the domain of digital circuits, for determining the global traversal time of a signal from the knowledge of bounds on the component propagation delays. The uncertainty on each component delay is given under the form of an interval. In [EF08], the authors focus on the following *inverse problem* for timing constraint graphs [CSD97]: "find intervals for component delays for which the global traversal time is guaranteed to be no greater than a specified maximum". The authors then introduce a method, the so-called *inverse method*, and show that this method solves the inverse problem in polynomial time.

The underlying principle of the inverse method may also be applied to other formalisms. Beside the extension to the probabilistic framework discussed in Chapter 6, other domains of applications of the inverse method are considered in Chapter 7, viz., directed weighted graphs and Markov decision processes. Research work related more specifically to those two frameworks are discussed at the end of Chapter 7.

Time-Abstract Bisimulation. The notion of time-abstract bisimulation, where one abstracts away both from the internal actions and from the time-elapsing, has been proposed in [LY93] in the context of a real-time process algebra. This notion of time-abstract bisimulation was then used in [TY01] in the framework of Timed Automata. Minimization algorithms for the region graph have also been proposed for Timed Automata in [ACH$^+$92, YL97] using bisimulations.

Synthesis of Parameters for Parametric Timed Automata. Parametric model-checking can be used to synthesize a constraint on the parameters such that a given property is verified.

The parameter design problem for timed automata (and more generally, for linear hybrid automata) was formulated and solved by Henzinger et al. in [HWT96], where a straightforward solution is given, based on the generation of the whole parametric state space until a fixpoint is reached. Unfortunately, in all but the most simple cases, this is is prohibitively expensive due, in particular, to the brute force exploration of the whole parametric state space.

The synthesis of constraints has been implemented in the context of PTAs or hybrid systems, e.g., in [AAB00] using tool TREX [CS01], or in [HRSV02] using an extension of UPPAAL [LPY97] for linear parametric model checking. Note that [AAB00] is able to infer non-linear constraints. Another interesting related work on PTA is presented in [HRSV02], which gives decidability results for the verification of a special class, called "L/U automata". Two subclasses of

L/U automata, called lower-bound and upper-bound PTA, are also considered in [WY03], with decidability results.

The problem of parameter synthesis for timed automata has been applied in particular to two main domains: telecommunication protocols and asynchronous circuits. For example, concerning telecommunication protocols, the Bounded Retransmission Protocol has been verified in [DKRT97] using UPPAAL [LPY97] and Spin [Hol03], and the Root Contention Protocol in [CS01] using TREX [ABS01]. The synthesis of constraints has also been studied more specifically in the context of asynchronous circuits, mainly by Myers and coworkers (see, e.g., [YKM02]), and by Clarisó and Cortadella (see, e.g., [CC05, CC07]), who have proposed methods with approximations. They also proceed by analyzing failure traces and generating timing constraints that prevent the occurrence of such failures. A difference with the work of [CC05, CC07] is that our method is an *exact* method. Also note that the approach proposed in [CPR08] allows to compute parametric regions guaranteeing a feasible schedule in the domain of schedulability analysis.

In [FJK08], the authors propose an extension based on the *counterexample guided abstraction refinement* (CEGAR) [CGJ$^+$00]. When finding a counterexample, the system obtains constraints on the parameters that *make* the counterexample infeasible. When all the counterexamples have been eliminated, the resulting constraints describe a set of parameters for which the system is safe.

The authors of [KP10] show how to synthesize a part of the set of all the parameter valuations under which a given property holds in a system modeled by a etwork of PTAs. This is done by using bounded model checking techniques applied to parametric timed automata. The central idea of this work is to unfold the computation tree of the considered model up to some depth, and then synthesize values for the parameters. As a consequence, this approach is limited by the fact that it is possible to synthesize parameters for existential properties only, actually properties specified in the existential part of CTL without the next operator (ECTL$_{-X}$). Our work differs to their work in the sense that the inverse method does not depend on a reference property but on a reference valuation.

When considering hybrid systems, an interesting approach allows in [AKRS08] to synthesize initial values for the variables of a linear hybrid system. Given an initial state and a “discrete-time trajectory” (which corresponds basically to our traces), their method synthesizes values for the system (hybrid) variables such that the behavior of the system starting from any of those values will be the same in term of “discrete-time trajectories”. Although the hybrid variables in linear hybrid systems are closer to our clocks (which are variables evolving with the time) rather than to our parameters (which are unknown constants), this work is interestingly linked to ours, because their method makes use of a reference valuation of the state variables and, as a consequence, can also be seen as an “inverse method”.

Chapter 4

Case Studies

– How would you account for this discrepancy between you and the twin 9000?
– Well, I don't think there is any question about it. It can only be attributable to human error.

2001: A Space Odyssey
(Stanley Kubrick)

In Chapter 3, we introduced the inverse method, allowing to synthesize parameters valuations guaranteeing the same behavior in term of trace sets as under a given parameter valuation. In this chapter, we show the practical interest of such a method. A first tool, IMITATOR was implemented under a Python script calling the HYTECH model checker. Because of various weaknesses due to HYTECH features and to the program itself, a second tool, IMITATOR II, was implemented in OCaml.

We already introduced a flip-flop circuit in Section 3.1.1, and synthesized a constraint guaranteeing its good behavior in Section 3.4. We consider here a range of case studies, asynchronous circuits and telecommunication protocols, and synthesize constraints for each of these case studies. We show for each example the interest of our method, by giving a criterion of robustness, or by optimizing some of the timing bounds of the system. We then compare the constraints synthesized by our method with constraints from the literature, when applicable. In particular, we apply our method to several abstractions of the SPSMALL memory sold by the chipset manufacturer ST-Microelectronics.

We give for each case study sufficient details to understand the model. For a fully detailed description, refer to [And10c].

Plan of the Chapter. We first present in Section 4.1 the two versions of the tool implementing the inverse method. We quickly introduce IMITATOR, and present more thoroughly IMITATOR II. In particular, we give implementation details and explain algorithmic optimizations.

We then present a range of case studies, i.e.:

- a sample example of "SR-latch" (Section 4.2),
- an "AND–OR" circuit (Section 4.3),
- the Root Contention Protocol (Section 4.4),
- the Bounded Retransmission Protocol (Section 4.5),
- an example of latch circuit (Section 4.6),
- various abstractions of a memory circuit built and sold by ST-Microelectronics (Section 4.7), and
- an example of networked automation system (Section 4.8).

We summarize the experiments in Section 4.9, and compare the computation times of IMITATOR and IMITATOR II. We finally compare in Section 4.10 the features offered by IMITATOR II with various other tools allowing to model and verify timed systems using timed automata or similar formalisms.

4.1 Tools

4.1.1 IMITATOR

The inverse method algorithm has first been implemented in the prototype IMITATOR [And09a], which is a script written in Python, driving the HYTECH [HHWT95] model checker for the computation of the *Post* operation. As a consequence, the input syntax is (almost exactly) the syntax of HYTECH. The Python program contains about 1500 lines of code, and it took about 4 man-months of work.

Various experiments have been conducted, allowing to synthesize parameters guaranteeing a "good" behavior (see [AEF09] for a summary).

Although IMITATOR allowed us to synthesize constraints on various case studies, the tool suffered from several limitations due to its interface with HYTECH. First, the arithmetics of HYTECH uses a limited precision, thus often leading to overflows. Second, it performs a static composition of the timed automata, thus preventing the designer from verifying more than a dozen of automata in parallel. Moreover, its fixpoint condition is as follows: the computation stops when all the states computed at some step are included (in the sense of constraint inclusion) in the states computed at previous steps. However, in the standard *IM* algorithm, the computation stops when all the states computed at some step are *equal* the states computed previously. As a consequence, the algorithm really implemented in IMITATOR was the variant $IM^{\subseteq}$ described in Section 3.5.1 rather than the standard inverse method *IM*.

All those reasons led to a new implementation of IMITATOR.

4.1.2 IMITATOR II

We introduce in this section the tool IMITATOR II [And10a].

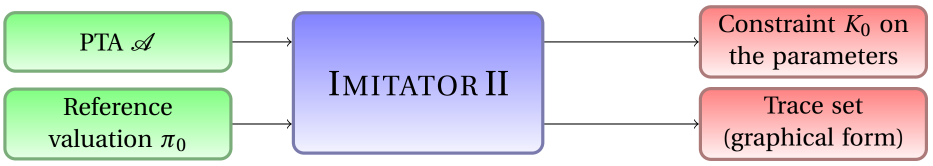

PTA $\mathscr{A}$
Reference valuation π_0
IMITATOR II
Constraint K_0 on the parameters
Trace set (graphical form)

Figure 4.1: IMITATOR II inputs and outputs in inverse method mode

Structure. Whereas IMITATOR was a prototype written in Python calling HYTECH for the computation of the *Post* operation, IMITATOR II is a standalone tool written in OCaml. The *Post* operation has been fully implemented, and the inverse method algorithm entirely rewritten. As depicted in Figure 4.2, IMITATOR II makes use of an external library for manipulating convex polyhedra. Depending on the user's preference, IMITATOR II can call either the NewPolka library, available in the APRON library [JM09], or the Parma Polyhedra Library (PPL) [BHZ08]. The trace sets are output under a graphical form using the DOT module of the graph visualization software Graphviz [gWp].

The syntax is close to the HYTECH syntax, with a few minor improvements. See [And10b] for the detailed syntax.

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

Internal Representation. States are represented using a triple (q, v, C) made of the current location q in each automaton, a value for each discrete variable[1] v, and a constraint C on the clocks and the parameters. In order to optimize the test of equality between a new computed state and the set of states

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

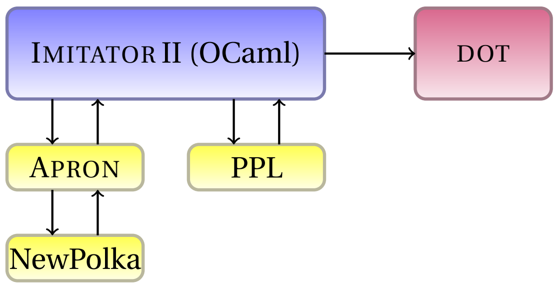


Figure 4.2: IMITATOR II internal structure

computed previously, the states are stored in a hash table as follows: to a given key (q, v) of the hash table, we associate a list of constraints $C_1, \ldots, C_n$, corresponding to the n states $(q, v, C_1), \ldots, (q, v, C_n)$.

Note that, like PHAVer but unlike HYTECH, IMITATOR II uses exact arithmetics with unlimited precision.

Contrarily to HYTECH which performs an *a priori* static composition of the automata, thus leading to a dramatical explosion of the number of locations, IMITATOR II performs an *on-the-fly* composition of the automata. This *on-the-fly* composition allows to analyze bigger systems, and decreases drastically the computation time compared to IMITATOR (see Section 4.9).

Features. IMITATOR II includes the following features:

- Full reachability analysis: given a PTA $\mathscr{A}$, compute the set of all the reachable states (as it is done in tools such as, e.g., HYTECH and PHAVer);
- Inverse method algorithm: given a PTA $\mathscr{A}$ and a reference valuation π_0, synthesize a constraint guaranteeing the same trace set as for $\mathscr{A}[\pi_0]$;
- Variant $IM^{\subseteq}$ of the inverse method algorithm (see Section 3.5.1);
- Automatic generation of the trace sets, for the reachability analysis and for the inverse method algorithm IM;
- Output of the trace sets under a graphical form using DOT. An example of trace set[2] automatically generated by IMITATOR II is given in Figure 4.3 under the form of an oriented graph, where nodes correspond to locations, and arrows correspond to transitions; note that locations of the same color are identical.

constants or other discrete variables. They can be used in guards and invariants, and can be compared with integers, clocks or parameters.

[2]This trace set actually corresponds to the trace set of the Bounded Retransmission Protocol, which will be studied in Section 4.5.

Note that IMITATOR II also features an implementation of the behavioral cartography algorithm, which will be described in Chapter 5.

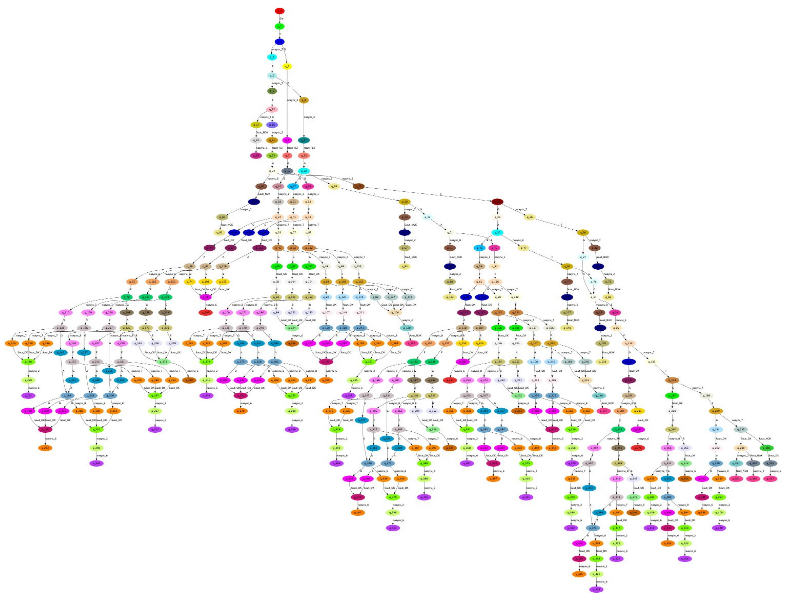


Figure 4.3: Example of trace set automatically output by IMITATOR II

Optimizations. The main optimization brought to the inverse method algorithm is the replacement of line 7 in Algorithm 1 by the portion of algorithm given in Algorithm 4.

Algorithm 4: Optimization of *IM* for IMITATOR II

```
1 for $(q, C) \in S$ do
2     $C \leftarrow C \wedge \neg J$
```

Line 7 in Algorithm 1 corresponds to the computation of all the states reachable in up to i steps from the initial state, with the new constraint K that has just been updated with the addition of some $\neg J$. However, this computation is redundant because we can show that:

- no new state can be computed (because K has been restrained with $\neg J$), and

- no state previously computed can be removed (because both $\neg J$ and the states previously computed are π-compatible).

As a consequence, it is safe to remove line 7 in Algorithm 1, thus avoiding a costly computation. Instead, we simply update the set S of states by adding $\neg J$ to all the states computed, as shown in Algorithm 4.

Algorithm 5: Simpler way to describe $IM(\mathcal{A}, \pi)$

```
input : A PTA $\mathcal{A}$ of initial state $s_0$
input : Valuation $\pi$ of the parameters
output: Constraint $K_0$ on the parameters

$S \leftarrow \{s_0\}$
while true do
    while there are $\pi$-incompatible states in $S$ do
        Select a $\pi$-incompatible state $(q, C)$ of $S$ (i.e., s.t. $\pi \not\models C$) ;
        Select a $\pi$-incompatible $J$ in $(\exists X : C)$ (i.e., s.t. $\pi \not\models J$) ;
        for $(q, C) \in S$ do
            $C \leftarrow C \land \neg J$
    if $Post_{\mathcal{A}(K)}(S) \sqsubseteq S$ then
        return $\bigcap_{(q,C)\in S}(\exists X : C)$
    $S \leftarrow S \cup Post_{\mathcal{A}(K)}(S)$
```

We present this simpler way of describing the inverse method in Algorithm 5. As a consequence of this simplification, we can save the use of variables i and K. Note also that the computation of the set $Post_{\mathcal{A}(K)}(S)$ is still redundant in Algorithm 5 because it is performed twice in this algorithm for the sake of clarity. Of course, only one such computation is performed in our implementation.

We also considered the following optimization: in order to reduce the size of the constraints in the memory, we factorize the constraint K as follows. Instead of adding $\neg J$ to each state of S, we only add $\neg J$ to K, and keep this constraint K separate. We add K to the constraint associated with a new state only when performing the equality test between this new constraint, and the constraints associated with the previous states, in which case we have to add K to both sides of the equality. Unfortunately, for the set of benchmarks we considered, this modification did not bring a significant gain in term of memory space, and the computation time raised significantly because of the repeated addition of K to constraints. As a consequence, this optimization has been discarded in the current version of the tool.

Options. The most useful options available for IMITATOR II are explained in the following. See [And10b] for a full list.

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

However, even for acyclic systems, several (different) traces can pass by the same state. In such a case, if the `-acyclic` option is activated, IMITATOR II will compute *twice* the states after the state common to the two traces. As a consequence, activating this option might even lead to a decrease of speed in some cases.

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

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

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

`-post-limit <limit>` **(default:** `none`**)** Limits the number of iterations in the *Post* exploration, i.e., the depth of the traces.

`-sync-auto-detect` **(default:** `false`**)** IMITATOR II considers that all the automata declaring a given synchronization label must be able to synchronize all together, so that the synchronization can happen. By default, IMITATOR II considers that the synchronization labels declared in an automaton are those declared in the `synclabs` section. Therefore, if a synchronization label is declared but never used in (at least) one automaton, this label will never be synchronized in the execution[3].

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

`-time-limit <limit>` **(default: `none`)** Try to limit the execution time (the value `<limit>` is given in seconds). Note that, in the current version of IMITATOR II, the test of time limit is performed at the end of an iteration only (i.e., at the end of a given *Post* iteration).

Advantages. In practice, the computation time of the inverse method implemented in IMITATOR II is (almost) insensitive to the size of the constants. However, it is possible that verifying a timed concurrent system using an external model checker is sensitive to the size of the constants. As a consequence, for systems with large constants that one can hardly verify using external tools, it is interesting to run the inverse method implemented in IMITATOR II to get a constraint. Then, one can infer from this constraint much smaller values of the parameters having the same behavior as under the original (large) valuation. The verification of such a *rescaled* system can then be much quicker in practice. Note, however, that the inverse method is more sensitive to the number of clocks and parameters, to the number of iterations, and to the branching of the PTA.

4.2 SR-Latch

We consider in this section a SR "NOR" latch, which is one of the most fundamental latches. S and R stand for set and reset. This latch (described in, e.g., [HH07]) is depicted in Figure 4.4 left. This circuit is made of two "NOR" gates. There are two input signals R and S, and two output signals Q and $\overline{Q}$. The stored bit is present on the output Q.

The possible configurations of the latch are the following ones:

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

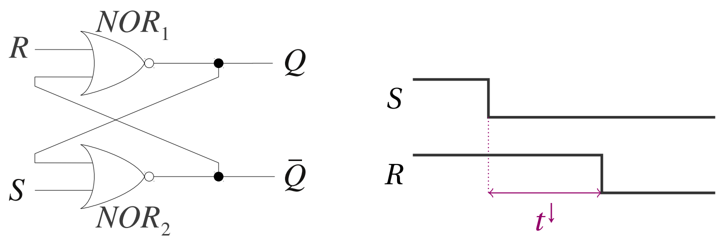


Figure 4.4: SR latch (left) and environment (right)

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

We consider an initial configuration with $R = S = 1$ and $Q = \overline{Q} = 0$. As depicted in Figure 4.4 right, the signal S first goes down. Then, the signal R goes down after a time $t^{\downarrow}$.

We consider that the gate NOR_1 (resp. NOR_2) has a punctual parametric delay δ_1 (resp. δ_2). Moreover, the parameter $t^{\downarrow}$ corresponds to the time duration between the fall of S and the fall of R.

Each location of the PTA $\mathscr{A}$ modeling this SR-latch corresponds to a different configuration of the signals R, S, Q and $\overline{Q}$. We give in Table 4.1 the correspondence between the name of the location q_i, for $i = 0, \ldots, 5$, and the values of the four signals (only the locations that are actually reachable from the initial state using our environment are depicted).

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

Table 4.1: Values of the signals for each of the locations of the SR-latch

We consider the following reference valuation π_0 of the parameters:

$$\delta_1 = 2 \quad \delta_2 = 2 \quad t^{\downarrow} = 1$$

Under π_0, it can be shown (e.g., using IMITATOR II in reachability mode) that the corresponding trace set is the one depicted in Figure 4.5.

$$\begin{tikzcd} q_0 \arrow[r, "S^{\downarrow}"] & q_1 \arrow[r, "R^{\downarrow}"] & q_2 \arrow[r, "\overline{Q}^{\uparrow}"] & q_4 \end{tikzcd}$$

Figure 4.5: Trace set for the SR latch under K_0

Synthesis of Parameters. Our goal is to synthesize a set of parameters guaranteeing the following good behavior: "the system always ends in a state where $\overline{Q} = 1$". This behavior corresponds to trace sets such that, for any trace of the trace set, the last location of the trace is such that $\overline{Q} = 1$. From Table 4.1, such locations are q_3 or q_4. One can see that the trace set of $\mathscr{A}[\pi_0]$, which is made of a single trace, satisfies this requirement, because the last location of the trace is q_4. As a consequence, $\mathscr{A}$ has a good behavior under π_0.

Let us now synthesize other parameters valuations corresponding to this behavior, by applying the inverse method to $\mathscr{A}$ and π_0. IMITATOR II synthesizes the following constraint K_0:

$$\delta_2 > t^{\downarrow} \ \wedge \ t^{\downarrow} + \delta_1 > \delta_2$$

From Theorem 3.8, the trace set corresponding to the system under any $\pi \models K_0$ is equal to the one given in Figure 4.5. It can be shown that this constraint K_0 is not maximal, i.e., there exist other parameters valuations having the same good behavior. It will be the purpose of Section 5.4.2 to synthesize the maximal constraint for this example.

4.3 AND–OR

This example deals with an "AND–OR" circuit described in [CC05] and depicted in Figure 4.6 (left). It is composed of 2 gates (one "AND" gate and one "OR" gate) which are interconnected in a cyclic way. The environment, depicted in Figure 4.6 (right), corresponds to 2 input signals a and b, with cyclic alternating rising edges and falling edges.

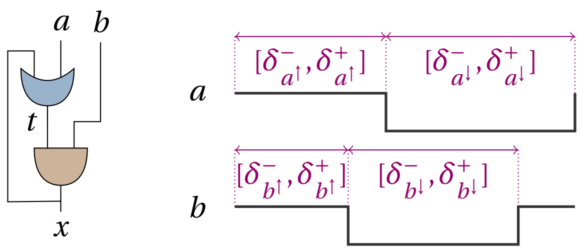

a
b
t
x
a
$[\delta^-_{a^{\uparrow}}, \delta^+_{a^{\uparrow}}]$
$[\delta^-_{a^{\downarrow}}, \delta^+_{a^{\downarrow}}]$
b
$[\delta^-_{b^{\uparrow}}, \delta^+_{b^{\uparrow}}]$
$[\delta^-_{b^{\downarrow}}, \delta^+_{b^{\downarrow}}]$

Figure 4.6: AND–OR circuit (left) and its environment (right)

Each rising (resp. falling) edge of signal a, is denoted by $a^{\uparrow}$ (resp. $a^{\downarrow}$), and similarly for b, t, x. The delay between the rising edge $a^{\uparrow}$ and the falling edge $a^{\downarrow}$ (resp. between $a^{\downarrow}$ and $a^{\uparrow}$) of signal a is in $[\delta^-_{a^\uparrow}, \delta^+_{a^\uparrow}]$ (resp. $[\delta^-_{a^\downarrow}, \delta^+_{a^\downarrow}]$), and similarly[4] for b. The traversal of the gate "OR" gate takes also a delay in $[\delta^-_{Or}, \delta^+_{Or}]$, and likewise for the "AND" gate. Those 12 timing parameters are bound by the following implicit constraint:

$$\begin{array}{lccccc} & \delta^-_{And} \leq \delta^+_{And} & \wedge & \delta^-_{Or} \leq \delta^+_{Or} & \wedge & \delta^-_{a^\downarrow} \leq \delta^+_{a^\downarrow} \\ \wedge & \delta^-_{a^\uparrow} \leq \delta^+_{a^\uparrow} & \wedge & \delta^-_{b^\downarrow} \leq \delta^+_{b^\downarrow} & \wedge & \delta^-_{b^\uparrow} \leq \delta^+_{b^\uparrow} \end{array}$$

Each of the 2 gates is modeled by a PTA, as well as the environment. We consider an inertial model for gates, where any change of the input may lead to a change of the output (after some delay). The PTA $\mathscr{A}$ modeling the system results from the composition of those 3 PTAs.

A bad state expresses the fact that the rising edge of output signal x occurs before the rising edge of a within the same cycle. We set the parameters to the following values, ensuring that the bad state is not reachable:

$$\begin{array}{cccc} \delta^-_{a^\uparrow} = 13 & \delta^+_{a^\uparrow} = 14 & \delta^-_{a^\downarrow} = 16 & \delta^+_{a^\downarrow} = 18 \\ \delta^-_{b^\uparrow} = 7 & \delta^+_{b^\uparrow} = 8 & \delta^-_{b^\downarrow} = 19 & \delta^+_{b^\downarrow} = 20 \\ \delta^-_{And} = 3 & \delta^+_{And} = 4 & \delta^-_{Or} = 1 & \delta^+_{Or} = 2 \end{array}$$

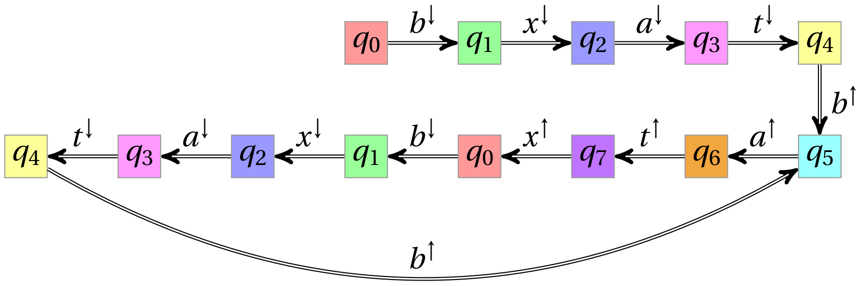
We consider an environment starting at location q_0 with $a = b = x = t = 1$, and the following repeated cycle of alternating rising and falling edges of a and b: $b^\downarrow, a^\downarrow, b^\uparrow, a^\uparrow$. For the given environment and the valuation π_0, the set of traces of the system is depicted in Figure 4.7 under the form of an oriented graph, where q_i, $0 \leq i \leq 7$, are locations of $\mathscr{A}$. The values of the signals of the system for each location q_i are given in Table 4.2. We can check that, in this graph, the bad state is not reached, i.e., the rising edges and falling edges of a, b, x alternate properly.

Using IMITATOR II applied to the PTA $\mathscr{A}$ modeling the system and the reference instantiation π_0, the following constraint K_0 is computed after 14 iterations:

$$\begin{array}{lccc} & \delta^-_{b^\downarrow} + \delta^-_{b^\uparrow} > \delta^+_{Or} + \delta^+_{a^\uparrow} & \wedge & \delta^-_{b^\uparrow} > \delta^+_{And} + \delta^+_{Or} \\ \wedge & \delta^+_{a^\downarrow} + \delta^+_{a^\uparrow} \geq \delta^-_{b^\downarrow} + \delta^-_{b^\uparrow} & \wedge & \delta^-_{a^\uparrow} > \delta^+_{And} + \delta^+_{b^\uparrow} \end{array}$$

Under any instantiation of the parameters $\pi \models K_0$, the set of traces under π is guaranteed to be identical to the set of traces under π_0 given in Figure 4.7 and, therefore, does not reach any bad state. In [CC05], the generated constraint is not given.

[4]Note however that the interval $[\delta^-_{b^\uparrow}; \delta^+_{b^\uparrow}]$ has a slightly different meaning, because it corresponds to the interval of delays between the rise of a and the fall of b, as shown in Figure 4.6 (right). This choice allows an easier modeling, and a more frequent termination of the analysis.



$$\begin{tikzcd}
 & & & & q_0 \arrow[r, "b^{\downarrow}"] & q_1 \arrow[r, "x^{\downarrow}"] & q_2 \arrow[r, "a^{\downarrow}"] & q_3 \arrow[r, "t^{\downarrow}"] & q_4 \arrow[d, "b^{\uparrow}"] \\
q_4 \arrow[rrrrrrrr, bend right, "b^{\uparrow}"'] & q_3 \arrow[l, "t^{\downarrow}"'] & q_2 \arrow[l, "a^{\downarrow}"'] & q_1 \arrow[l, "x^{\downarrow}"'] & q_0 \arrow[l, "b^{\downarrow}"'] & q_7 \arrow[l, "x^{\uparrow}"'] & q_6 \arrow[l, "t^{\uparrow}"'] & q_5 \arrow[l, "a^{\uparrow}"'] & q_5 \arrow[l, "a^{\uparrow}"']
\end{tikzcd}$$

Figure 4.7: Trace of the AND–OR circuit under π_0

Location	a	b	t	x
q_0	1	1	1	1
q_1	1	0	1	1
q_2	1	0	1	0
q_3	0	0	1	0
q_4	0	0	0	0
q_5	0	1	0	0
q_6	1	1	0	0
q_7	1	1	1	0

Table 4.2: Locations of the AND–OR circuit

This constraint gives a criterion of robustness for this system by guaranteeing that, for values of the parameters around the reference valuation, the system will still behave well. It is in particular interesting to note that several parameters do not appear in the constraint synthesized (and are actually only bound by the implicit constraint given earlier). This is the case of parameters $\delta^-_{a\downarrow}$, $\delta^+_{b\downarrow}$, δ^-_{And} and δ^-_{Or}. This means that, for the considered environment, the value of these parameters has no influence on the behavior of the system.

4.4 IEEE 1394 Root Contention Protocol

Description of the Model. This case study concerns the Root Contention Protocol of the IEEE 1394 ("FireWire") High Performance Serial Bus, considered in the parametric framework in [CS01, HRSV02], and in the probabilistic framework in [KNS03]. As described in [HRSV02], this protocol is part of a leader election protocol in the physical layer of the IEEE 1394 standard, which is used to break symmetry between two nodes contending to be the root of a tree, spanned in the network technology. The protocol consists in first drawing a random number (0 or 1), then waiting for some time according to the result drawn, followed by the sending of a message to the contending neighbor. This is repeated by both nodes until one of them receives a message before sending one, at which point the root is appointed.

We consider the following five timing parameters:

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

Those timing parameters are bound by the following implicit constraint:

$$f\_min \leq f\_max \quad \wedge \quad s\_min \leq s\_max$$

The model we consider is a nonprobabilistic version of the probabilistic given in [KNS03, pWpb], where the probabilistic distributions have been replaced with nondeterminism. We give in Figure 4.8 the PTA model of node i, and in Figure 4.9 the PTA model of wire i. We make use in Figure 4.8 of the notion of *urgent* locations: the semantics is that the time cannot pass inside these locations, and one must take a transition immediately after entering it. This is only syntactic sugar which is equivalent to the use of one more clock that is

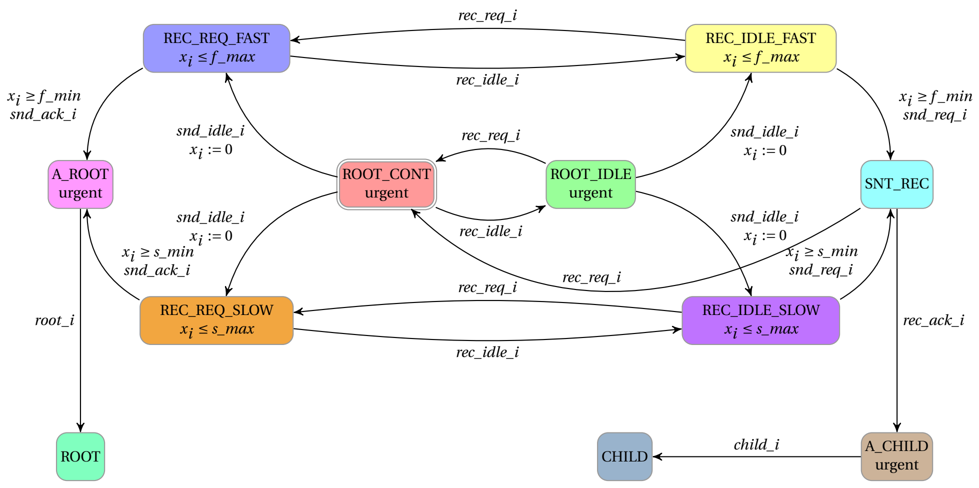


Figure 4.8: PTA modeling node i in the Root Contention Protocol

reset when entering the location, and that must be equal to 0 when leaving the location through any transition. Moreover, in both Figure 4.8 and Figure 4.9, we exceptionally integrate the invariant *in* the location, for the sake of readability. As usual, a location without any invariant is considered to have an invariant equal to `true`.

Synthesis of Constraints. We aim at synthesizing a constraint for the following reference valuation π_0, which corresponds to the IEEE standard with wire length near to the maximum possible according to [HRSV02] (timings are given in ns):

$$f\_min = 760 \qquad f\_max = 850 \qquad delay = 360$$
$$s\_min = 1590 \qquad s\_max = 1670$$

By applying IMITATOR II to this model and the reference valuation π_0, we synthesize the following constraint K_0:

$$s\_min > 2 * delay + f\_max \land delay \geq 0 \land f\_min > 2 * delay$$

Observe that this constraint is exactly the same as the one synthesized in [HRSV02]. This constraint is also very similar to the one synthesized in [CS01]; the only difference is that our constraint is larger, because we do not constraint *delay* to be strictly positive.

We give in Figure 4.10 the trace set of the protocol under any $\pi \models K_0$, as automatically output by IMITATOR II.

The main interest brought by the synthesis of this constraint is that it gives a criterion of robustness to the system. Similarly, it shows that this protocol is

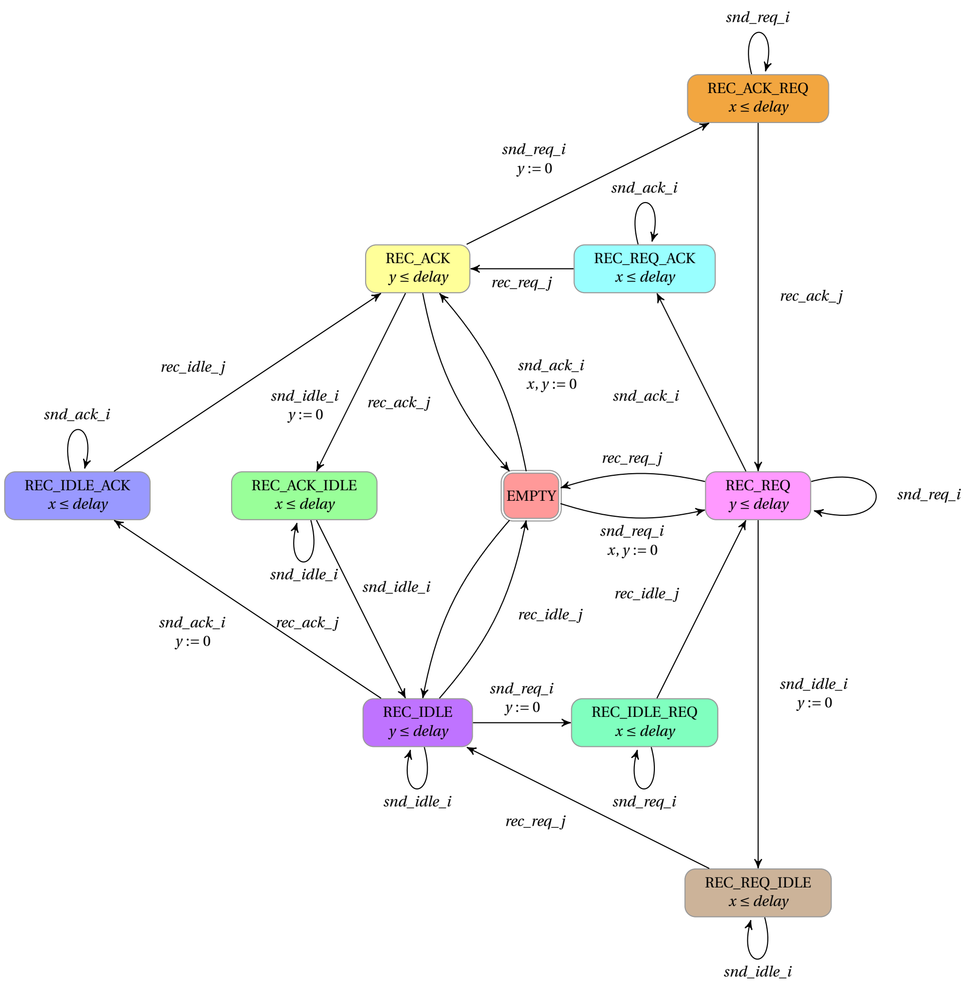


Figure 4.9: PTA modeling wire i in the Root Contention Protocol

also correct for values of the parameters other than the ones given by the IEEE reference.

We will enlarge in Section 6.6.2 this constraint K_0 by computing a *behavioral cartography* (see Chapter 5) of the Root Contention Protocol according to the 3 parameters *delay*, *s_min* and *s_max*. We will also see in Chapter 6 that this constraint becomes very interesting in the *probabilistic* framework, because it will preserve the values of probabilities, in a probabilistic version of this model.

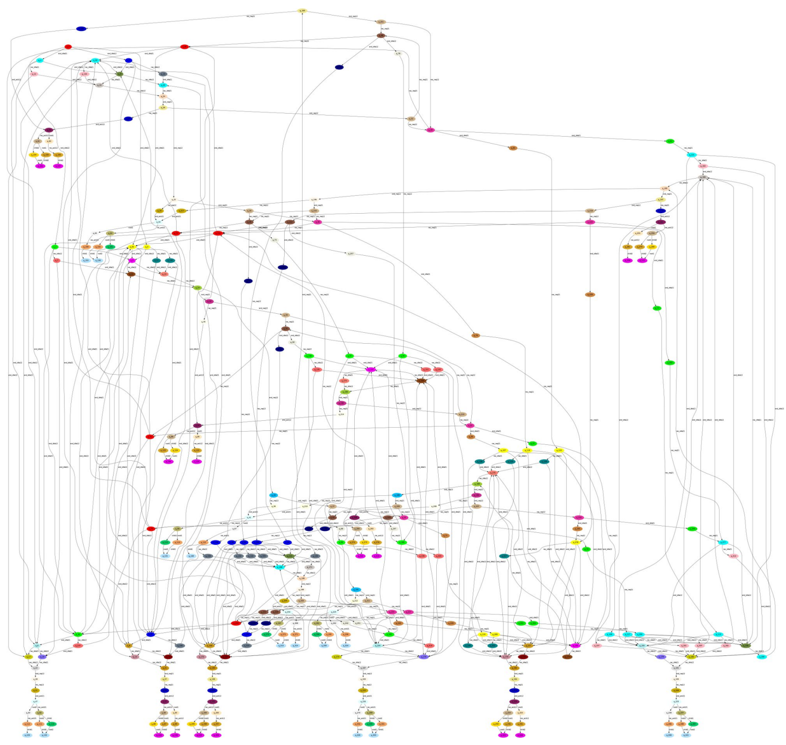


Figure 4.10: Trace set of the RCP output by IMITATOR II

4.5 Bounded Retransmission Protocol

We study here the Bounded Retransmission Protocol described and modeled using timed automata in [DKRT97]. As said in [DKRT97], this protocol, used in one of Philips' products, is based on the well-known alternating bit protocol but is restricted to a bounded number of retransmissions of a chunk, i.e., part of a file. So, eventual delivery is not guaranteed and the protocol may abort the file transfer. Timers are involved in order to detect the loss of chunks and the abortion of transmission.

The protocol consists of a sender equipped with a timer, and a receiver equipped with another timer, which exchange data via two unreliable (lossy) channels.

The model considered here is a slightly simplified version of the model of [DKRT97]. In particular, a loop in the model of the sender has been discarded, implying the fact that the sender tries to send only one file.

As in [DKRT97], we consider the following five timing parameters for the model.

- N stands for the number of chunks of a file;

- *SYNC* corresponds to the delay added after a failure in order to assure that the sender does not start transmitting a new file before the receiver has properly reacted to the failure;

- *T1* corresponds to the time-out of the sender for initiating a retransmission when the sender has not received an acknowledgment from the receiver;

- *TR* corresponds to the time-out of the receiver for indicating failure when it has not received the last chunk of a file

- *TD* corresponds to the maximum delay in communication channels.

We consider the following valuation π_0 of the parameters of the system:

$$\begin{array}{rrr} MAX = 2 & N = 2 & TD = 1 \\ T1 = 3 & TR = 16 & SYNC = 17 \end{array}$$

We consider a slightly simplified model of the protocol, where the system stops when one file has been successfully sent. Using IMITATOR II applied to the PTA $\mathscr{A}$ modeling the system and the reference valuation π_0, the following constraint K_0 is computed:

$$\begin{array}{lclcl} & N = 2 & \wedge & MAX = 2 \\ \wedge & TR + TD > 5 * T1 & \wedge & TR \geq 3 * TD + 4 * T1 \\ \wedge & 2 * TD + 5 * T1 > TR & \wedge & SYNC + TD \geq TR \\ \wedge & T1 > 2 * TD \end{array}$$

Recall that the corresponding trace set, automatically generated by IMITATOR II, was given in Figure 4.3 page 83. As for the Root Contention Protocol, the main interest brought by the synthesis of this constraint is that it gives a criterion of robustness to the system. Similarly, it shows that this protocol is also correct for values of the parameters other than the ones given by the reference valuation π_0.

Comparison with other methods. In [DKRT97], the authors synthesize the following (non-linear) constraints guaranteeing that (1) premature time-outs are not possible, and (2) sender and receiver resynchronize after an abort.

$$Z: \; T1 > 2 * TD \; \wedge \; SYNC \geq TR \geq 2 * MAX * T1 + 3 * TD$$

Note that, since $\pi_0 \models Z$, our constraint K_0 also guarantees that the assumptions of [DKRT97] are satisfied. It can be shown that our constraint K_0 is incomparable with this constraint Z.

The analysis of the complete model of the protocol (as described in [DKRT97]) has also been considered in [Sou10b] using the variant $IM^{\subseteq}$ implemented in IMITATOR II. The following constraint $K^{\subseteq}$ is computed using the same reference valuation π_0:

$$\begin{array}{rrcr} & N = 2 & \wedge & MAX = 2 \\ \wedge & TR + TD \geq 5 * T1 & \wedge & TR \geq 3 * TD + 4 * T1 \\ \wedge & 2 * TD + 5 * T1 \geq TR & \wedge & T1 \geq 2 * TD \\ \wedge & SYNC + T1 \geq TR + TD & & \end{array}$$

It can be shown that this constraint $K^{\subseteq}$ is incomparable both with our constraint K_0, and with the constraint Z of [DKRT97].

4.6 Latch Circuit

We consider in this section a latch circuit studied in the case of ANR project VALMEM (see Section 4.7 for a description of the project). This circuit, depicted in Figure 4.11, contains 5 elements: 2 "NOT" gates (viz., Not_1 and Not_2), one "XOR" gate (viz., element Xor), one "NAND" gate (viz., element And), as well as one "latch" element (viz., element $Latch$).

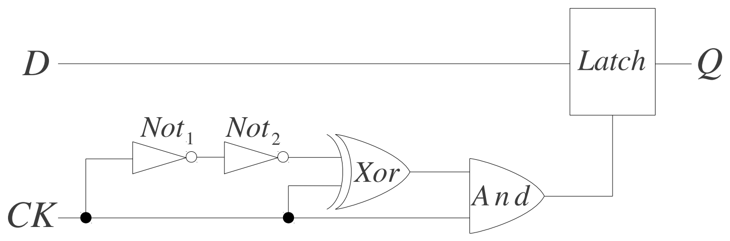


Figure 4.11: A latch circuit

Each of the four gates has a constant delay for a change of an input leading to a rising edge in the output, and another constant delay for a change of an input leading to a falling edge of the input. For example, when the input of the gate Not_1 is equal to 0 and raises, then the output will change after delay $\delta_{Not1^{\uparrow}}$. If the input is equal to 1 and falls, the delay before the output changes is $\delta_{Not1^{\downarrow}}$, and similarly for the other 3 gates. The latch has a single constant delay $\delta_{Latch^{\uparrow}}$ corresponding to the time needed between a change of its inputs and the raise of Q. There are 4 other parameters (viz., $T_{HI}, T_{LO}, T_{Setup}$, and T_{Hold}) used to model the environment. The rising (resp. falling) edge of signal D is denoted by $D^{\uparrow}$ (resp. $D^{\downarrow}$), and similarly for signals CK and Q.

We consider an environment starting from $D = CK = Q = 0$, with the following ordered sequence of actions for inputs D and CK: $D^{\uparrow}$, $CK^{\uparrow}$, $D^{\downarrow}$, $CK^{\downarrow}$, as depicted in Figure 4.12. Therefore, we have the implicit constraint $T_{Setup} \leq T_{LO} \wedge T_{Hold} \leq T_{HI}$.

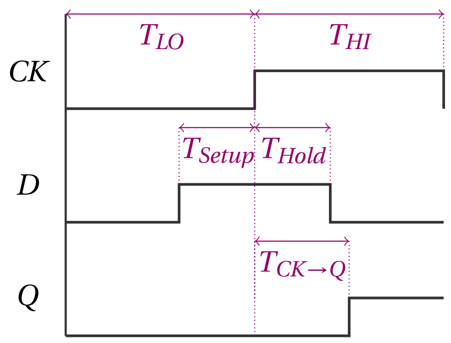


Figure 4.12: Environment for the latch circuit

Each element is modeled by a PTA, as well as the environment. The PTA $\mathscr{A}$ modeling the system results from the composition of those 6 PTAs.

The following valuation π_0 of the 13 parameters (in ps) was extracted from the circuit description by simulation computed in the VALMEM project:

$T_{HI} = 1000$	$T_{LO} = 1000$	$T_{Hold} = 350$	$T_{Setup} = 1$
$\delta_{Not1^{\uparrow}} = 219$	$\delta_{Not1^{\downarrow}} = 147$	$\delta_{Not2^{\uparrow}} = 155$	$\delta_{Not2^{\downarrow}} = 163$
$\delta_{Xor^{\uparrow}} = 147$	$\delta_{Xor^{\downarrow}} = 416$	$\delta_{And^{\uparrow}} = 80$	$\delta_{And^{\downarrow}} = 155$
$\delta_{Latch^{\uparrow}} = 240$			

A bad state corresponds to the fact that the output signal Q has not changed before the end of the cycle of signal CK. Under this valuation, we can show that the system does not reach any bad state.

Applying IMITATOR II to the PTA version of this model and the reference valuation π_0, the following constraint K_0 is synthesized:

$$\begin{array}{lc}
\wedge & \delta_{Xor^{\downarrow}} + \delta_{Not2^{\uparrow}} + \delta_{Not1^{\downarrow}} > T_{Hold} \\
\wedge & \delta_{Latch^{\uparrow}} + \delta_{And^{\uparrow}} > \delta_{Not2^{\uparrow}} + \delta_{Not1^{\downarrow}} \\
\wedge & \delta_{Not1^{\downarrow}} > \delta_{And^{\uparrow}} \\
\wedge & T_{Hold} > \delta_{Latch^{\uparrow}} + \delta_{And^{\uparrow}} \\
\wedge & T_{LO} \geq T_{Setup} \\
\wedge & T_{HI} \geq \delta_{And^{\downarrow}} + \delta_{Xor^{\downarrow}} + \delta_{Not2^{\uparrow}} + \delta_{Not1^{\downarrow}}
\end{array}$$

Under this constraint, the system has the same behavior as under the reference valuation, and therefore guarantees a good behavior of the system. Suppose now that we are interested in minimizing the T_{Hold} value, provided the system keeps its good behavior. Such a minimization has the following practical interest: since T_{Hold} corresponds to the duration during which the input signal D should be stable before the rise of the clock, minimizing this value allows to integer this portion of circuit in a *faster* environment. By instantiating all the parameters in K_0 with their value in π_0, except T_{Hold}, we get the following constraint:

$$320 < T_{Hold} < 718$$

So we can minimize the value of T_{Hold} to 321, and we guarantee that the system will have exactly the same behavior as before.

4.7 SPSMALL Memory

We consider in this section the SPSMALL memory, which is a memory circuit sold by ST-Microelectronics. This memory has been first studied in the MEDEA+BLUEBERRIES (T126) European project involving ST-Microelectronics and the LSV laboratory (École Normale Supérieure de Cachan). It was then studied in the ANR VALMEM project involving, besides ST-Microelectronics and LSV, the LIP 6 laboratory (Université Pierre et Marie Curie).

4.7.1 Description

The SPSMALL memory actually corresponds to a *class* of small memories with a maximum total capacity of 64 kbits. Each instance of the memory is built by a parametrized compiler, where the number of words and the size of the words are parameters[5]. The number of words is ranking from 3 to 512 words, and the number of bits from 2 to 256 bits. We consider throughout this section the smallest memory consisting in 3 words of 2 bits (or abstractions of it), which leads to a netlist of 305 transistors.

The SPSMALL is manually built directly at the transistor level. Indeed, in order to be able to optimize the memory array part of the circuit, one must tune it manually. Moreover, the control logic and the decoder logic uses hand-made cells, and these complex structures cannot be automatically generated.

Our Approach. Before describing the memory and the model considered in this section, we first give the methodology used in the VALMEM project.

As depicted in Figure 4.13, a description of the memory under the form of a transistor netlist is given by ST-Microelectronics. Then, a functional abstraction generates a description of the memory in the functional description language VHDL, using automatic techniques developed by LIP 6 [SRD09]. At the

[5]This notion of parameter is not anyhow linked to the *timing parameters* (set P) mentioned throughout this thesis. We will study this memory for a given instance of these words and size parameters, and the parameters that we will consider correspond to internal timing delays, as for the other case studies.

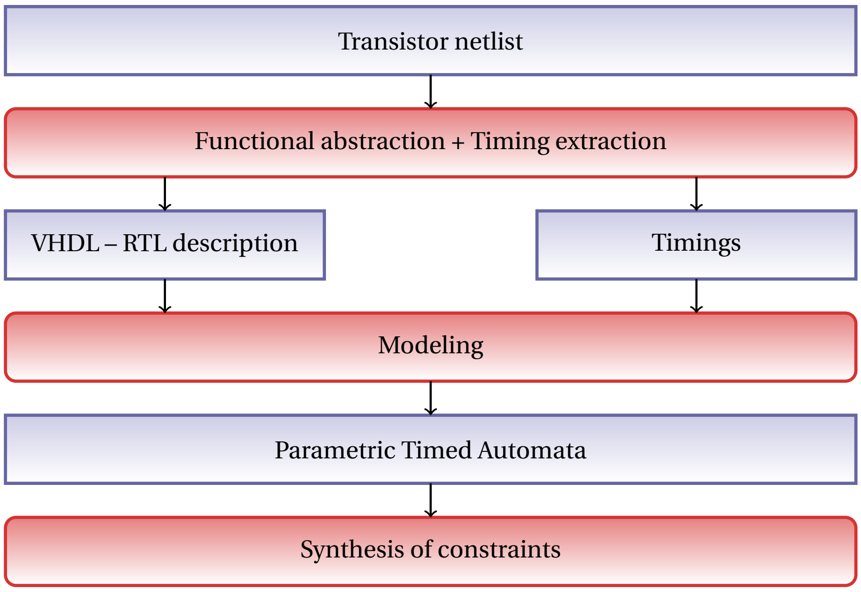


Figure 4.13: Methodology of the VALMEM project

same time, timings are extracted under the form of traversal delays of the elements. The next step is the translation by LIP 6 of the VHDL code into a network of (instantiated) timed automata, using the tool VHDL2TA [Bar09] developed in the framework of the VALMEM project. Finally, using a parametric version of those timed automata, and the reference instantiation of the parameters, we synthesize using IMITATOR II constraints guaranteeing a good behavior for the memory. Although we will mostly focus on this latter task in this section, we recall various information on the global process for the sake of understanding.

Level of Modeling. We borrow part of the following description to [BC05, CEFX09]. A memory circuit aims at storing data at some addressed locations, and is associated with two operations: *write* and *read*. A memory can be modeled at different levels of complexity, e.g., in an increasing order: at the functional block level, at the "latch" level, at the gate level, or at the transistor level. For the SPSMALL memory, the model can thus be implemented using 3 main components at the block level (see [BC05]), a few dozens of components at the latch level, about one hundred components at the gate level, or 305 components at the transistor level. There is a tradeoff in finding the appropriate level of modeling. The lower the level of modeling is, the more faithful to the reality the model is, but the more difficult the verification process is. In [CEFX09] and in this section, we choose to represent the memory at the latch level. The advantage is to limit the number of components at a reasonable size, and to have a “schematics” describing the architecture of the memory at this level, which closely corresponds to the VHDL code automatically produced.

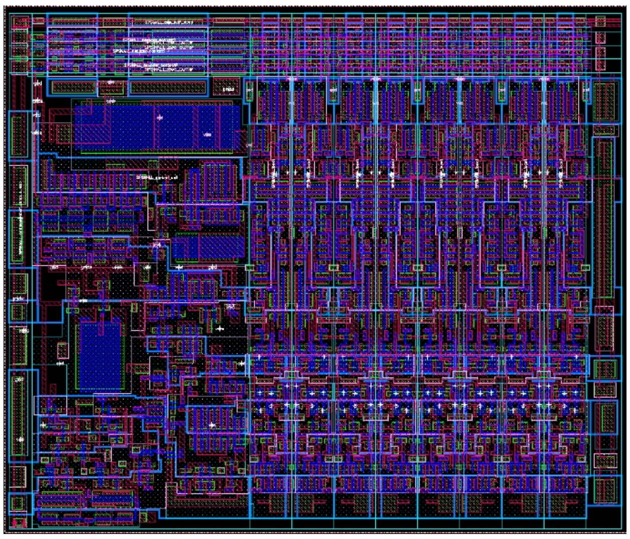


Figure 4.14: Transistor representation of the SPSMALL memory

In order to better illustrate the complexity of this memory, we give in Figure 4.14 a graphical representation of the memory at the transistor level.

Inputs and Outputs. The SPSMALL memory circuit has several input ports and one output port. The signals driven by input ports are:

- CK, the signal of the periodic clock;
- D, the n-bit width signal representing the data to be stored;
- A, the $\log_2(m)$-bit width signal representing the address of an internal memory location;
- WEN, the 1-bit width signal representing either a write or a read operation.

The signal driven by the output port is Q (of n-bit width). The data are stored in a memory array composed of $m \times n$ memory points. A memory location is a collection of n memory points. The write operation ($WEN = 0$ when CK is rising) writes the value of D in the internal memory location selected by A, and propagates D on output port Q. Such a memory is called a *write-through* memory. The read operation ($WEN = 1$ when CK is rising) outputs on port Q a copy of the data stored in the memory location selected by A.

Timing Parameters. We consider here the *write* operation. The environment for this operation is depicted in Figure 4.15.

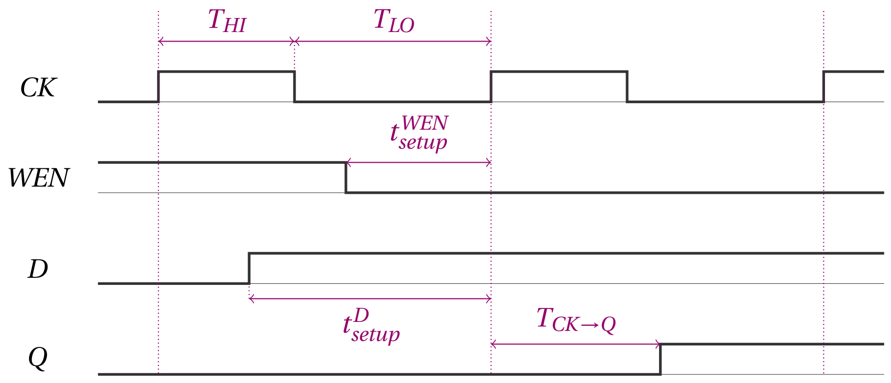


Figure 4.15: Environment for the *write* operation of SPSMALL

The duration of the clock cycle is parameterized by T_{HI} (duration of the high edge) and T_{LO} (duration of the low edge). We study this operation for two clock cycles. The parameter t_{setup}^{WEN} corresponds to the time during which the *WEN* signal should be stable before the beginning of the second clock cycle, i.e., the second rise of *CK*. Similarly, the parameter t_{setup}^{D} corresponds to the time during which the *D* signal should be stable before the beginning of the second clock cycle. Finally, the parameter $T_{CK\to Q}$ corresponds to the maximal time between the beginning of the second clock cycle and the rise of the output signal *Q*. Besides these 5 parameters, the SPSMALL memory is characterized by other parameters corresponding to the traversal delays of the gates and latches of the circuit.

Each of those parameters is given a valuation. Parameters valuations corresponding to the environment (viz., T_{HI}, T_{LO}, t_{setup}^{WEN}, t_{setup}^{D}) are taken from the datasheet of the memory given by ST-Microelectronics. Parameters valuations corresponding to internal delays are synthesized as follows. In the BLUEBERRIES project, they were manually computed by electrical simulation for a single configuration of the environment. In the VALMEM project, there are automatically retrieved using the transistor netlist (see Figure 4.13 page 99): from all possible inputs and outputs for a given component, only two values are kept, namely the lower and the upper bounds of the traversal time taken on all the possible configurations. Although this gives suitable results for the gates, the bounds are sometimes far from each other for memory points, thus weakening the precision of the verification. Those internal delays depend on the size of the transistors and on the technology used. This is the value of those internal delays which induces the possible values of the environment parameters.

Actually, this memory circuit has *two* different implementations for the same architecture. In other words, for the same schematics of gates and latches, there are two different sets of valuations of the parameters (i.e., environment parameters and traversal delays). The first implementation (SP1) corresponds to a fast component with a high power consumption, whereas the second implementation (SP2) corresponds to a slower component with a lower power consumption.

4.7.2 A Short History

The SPSMALL memory was first studied in [BC05], where the authors verify this memory component modeled by timed automata, using the real-time model checkers HyTech and Uppaal. In particular, the authors take into account the electrical propagation delays through gates and along wires. The authors propose an abstraction of the memory sufficiently small to be (manually) described in the model-checker Uppaal. Then they verify that, for some internal timings given by ST-Microelectronics, the read and write access timings are correct. Moreover, they verify that those access timings (viz., $T_{CK \rightarrow Q}$ for the write operation) are *optimal* by showing that the memory model has correct behaviors with those timings, whereas incorrect behaviors occur when choosing smaller timings. This is done by manually decreasing those timings, and checking that the behavior remains correct. Note that the authors consider here only *integer* timings, and do not investigate what is happening between two integer values.

The SPSMALL memory was then studied in [CEFX06], where the authors propose a high-level formalism, called Abstract Functional and Timing Graph (AFTG), for describing the memory. This formalism allows in particular to combine logical functionality and timing. After translation of the AFTG into the form a timed automaton, the authors are able to compute the response times of the modeled memory, and check their consistency with the values specified in the datasheet. The authors then go one step further by showing not only that the access timings are correct, but they also give the *optimal* input setup and hold timings such that the access timings remain correct. This is done by manually decreasing those input timings, and check that the access timings remain correct. Note that the authors also consider here only *integer* timings, and do not investigate what is happening between two integer values.

In [CEFX09], the authors then manually synthesize constraints on the setup and internal timings seen as *parameters* guaranteeing that the response times to a write command or a read command lie between certain bounds. Those constraints, derived using the SP1 implementation of the memory, can be immediately applied to other instances of the parameters to verify the behavior of other versions of the memory, such as SP2. Contrarily to the first two approaches, this work allows to consider dense (i.e., real) values for the timings, and give a criterion of robustness to the timings of the memory.

Our aim is to *automatically* derive constraints on the internal timings seen as parameters, such that the memory behaves well. We study in the forthcoming sections several abstractions of the SPSMALL memory.

4.7.3 Manually Abstracted Model

Description. We consider here a model manually abstracted, close to the model considered in [CEFX06]. We recall the model considered in [CEFX06] in Figure 4.16 under the form of an AFTG. This model was abstracted in order to consider that only one bit is stored. As a consequence, *D* becomes a 1-bit signal. Furthermore, we consider only the portion of the circuit relevant to the *write* operation.

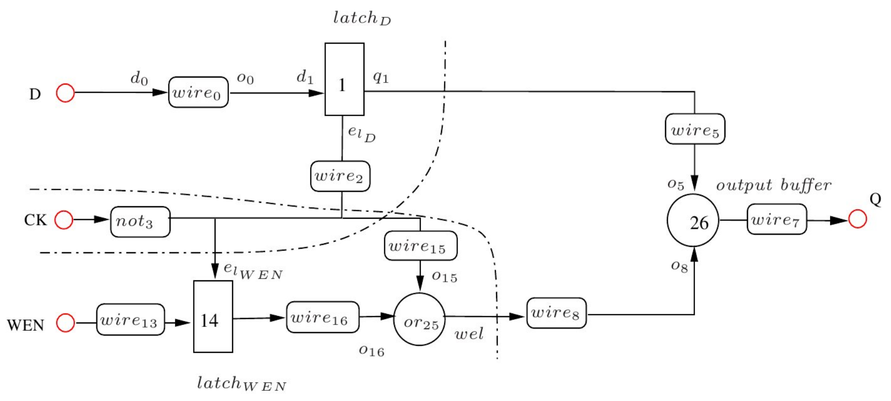


Figure 4.16: Abstract model of the SPSMALL memory (write operation)

Although the model we consider here is close to the model considered in [CEFX06], a major difference with the model of [CEFX06] though is that delays are not only associated with latches and wires anymore, but with latches, wires and *gates*, depending on the components. This model has been designed partially automatically from the VHDL code, using abstractions. This VHDL source code (available in [vWp]) was itself manually written.

This model, depicted in Figure 4.17, results in 9 components. Components $delay_D$ and $delay_{WEN}$ are delays (i.e., the logical functionality is the identity), components NOT_1, NOT_2 and NOT_3 are "NOT" gates, *WEL* is an "OR" gate, and components $delay_{WEN}$, $latch_D$ and net_{27} are latches. A further difference with the model considered in [CEFX06] is that several components have been grouped together in order to avoid the state-space explosion problem[6]. For example, several delays associated with wires have been incorporated into the previous elements: this is the case, e.g., of component $wire_5$ from Figure 4.16, the delay of which has been incorporated into the element $latch_D$, resulting in only one component ($latch_D$) in our model depicted in Figure 4.17.

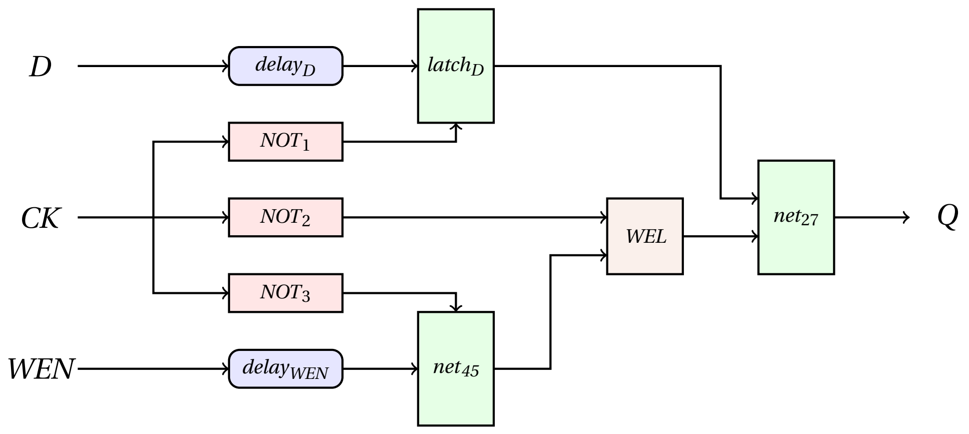


Figure 4.17: PTAs modeling the write operation of SPSMALL

Each of the components depicted in Figure 4.17 (wires, gates, latches) is modeled using a PTA. The translation of the gates into PTAs has been performed automatically using a preliminary version of VHDL2TA. The other components were manually written, and so was the composition of all components together. The environment is also modeled using a PTA. This results in a model containing 10 automata, 10 clocks and 26 parameters corresponding to the traversal delays of the components and the environment. Contrary to [CEFX06], the PTAs modeling the gates are actually *complete*, in the sense that all possible configurations and transitions are modeled, not only the configurations that will be met for a precise environment, as it was the case in [CEFX06]. This is in particular due to the automatic generation of the PTAs.

Implementation SP1. We give below the set of parameters valuations (say, π_1) coming from the implementation SP1 and adapted to this first model (timings are given in tens of pico-seconds).

[6]This model was actually first designed to be analyzed using HYTECH, which can hardly accept more than 10 components modeled by PTAs in parallel. However, analyzing this model using IMITATOR II is performed easily in a couple of seconds.

$$\begin{array}{rrr}
d\_up\_q\_0 = 21 & d\_dn\_q\_0 = 20 & d\_up\_net27 = 0 \\
d\_dn\_net27 = 0 & d\_up\_d\_inta = 22 & d\_dn\_d\_inta = 45 \\
d\_up\_wela = 0 & d\_dn\_wela = 22 & d\_up\_net45a = 5 \\
d\_dn\_net45a = 4 & d\_up\_net13a = 19 & d\_dn\_net13a = 13 \\
d\_up\_net45 = 21 & d\_dn\_net45 = 22 & d\_up\_d\_int = 14 \\
d\_dn\_d\_int = 18 & d\_up\_en\_latchd = 28 & d\_dn\_en\_latchd = 32 \\
d\_up\_en\_latchwen = 5 & d\_dn\_en\_latchwen = 4 & d\_up\_wen\_h = 11 \\
d\_dn\_wen\_h = 8 & d\_up\_d\_h = 95 & d\_dn\_d\_h = 66 \\
T_{HI} = 45 & T_{LO} = 65 & t^{D}_{setup} = 108 \\
t^{WEN}_{setup} = 48 & &
\end{array}$$

Constraint. Applying IMITATOR II to this model and the reference valuation π_1 (corresponding to the SP1 implementation), one synthesizes the following constraint K_1 after 32 iterations (31 reachable states with 30 transitions):

$$\begin{array}{ll}
 & T_{HI} + d\_up\_net13a > d\_dn\_net13a + d\_dn\_wela + d\_up\_net27 + d\_up\_q\_0 \\
\land & T_{LO} > d\_up\_en\_latchd + d\_up\_d\_int + d\_up\_d\_inta \\
\land & t^{D}_{setup} + d\_dn\_en\_latchd > d\_up\_d\_h + d\_up\_d\_int + d\_up\_d\_inta \\
\land & t^{WEN}_{setup} + d\_up\_d\_h > t^{D}_{setup} + d\_dn\_wen\_h + d\_dn\_net45 + d\_dn\_net45a + d\_up\_wela \\
\land & T_{LO} + d\_dn\_wen\_h > t^{WEN}_{setup} + d\_up\_net13a + d\_up\_wela \\
\land & T_{HI} > d\_dn\_net13a + d\_dn\_wela \\
\land & T_{LO} > t^{WEN}_{setup} + d\_up\_en\_latchwen \\
\land & t^{D}_{setup} > d\_up\_d\_h \\
\land & t^{D}_{setup} \geq T_{LO} \\
\land & T_{LO} + T_{HI} \geq t^{D}_{setup} \\
\land & d\_dn\_en\_latchwen \geq 0 \\
\land & d\_up\_en\_latchwen \geq 0 \\
\land & t^{WEN}_{setup} + d\_up\_en\_latchd > T_{LO} + d\_dn\_wen\_h \\
\land & d\_dn\_net13a > d\_dn\_en\_latchwen \\
\land & t^{WEN}_{setup} + d\_up\_net13a > T_{LO} \\
\land & d\_up\_en\_latchwen + d\_up\_net45 + d\_up\_net45a > d\_up\_en\_latchd \\
\land & d\_dn\_net13a + d\_dn\_wela > d\_dn\_en\_latchd \\
\land & d\_up\_wela \geq 0 \\
\land & t^{D}_{setup} + d\_up\_en\_latchd + d\_dn\_d\_int + d\_dn\_d\_inta > T_{LO} + d\_up\_d\_h \\
\land & d\_up\_en\_latchd + d\_up\_d\_int + d\_up\_d\_inta > d\_up\_en\_latchwen + d\_dn\_net45 + d\_dn\_net45a \\
\land & d\_up\_d\_h + d\_up\_d\_int + d\_up\_d\_inta > t^{D}_{setup} + d\_dn\_net13a \\
\land & d\_dn\_net13a + d\_dn\_wela + d\_up\_net27 + d\_up\_q\_0 > T_{HI} + d\_up\_en\_latchwen
\end{array}$$

Interpretation. The main advantage of the constraint synthesized by IMITATOR II is that it allows to show the link between the internal timing delays and the external values of the environment. Indeed, the timing parameters corresponding to the environment are *constrained* by the internal traversal delays of the gates, wires and latches. Despite the complex form of the constraint synthesized, it is possible to give an interpretation for some of the inequalities.

First of all, some inequalities are actually synthesized because of the environment that we consider. Inequalities such as $t^{D}_{setup} \geq T_{LO}$ or $T_{LO} + T_{HI} \geq t^{D}_{setup}$ come from the way we modeled the environment, and are bound by the *model* more than the system.

Moreover, other inequalities can be interpreted as a guarantee on the *order* of the events. Recall that our inverse method guarantees the same trace sets and, as a consequence, the same ordering of events. For example, the inequality $t^{WEN}_{setup} + d\_up\_en\_latchd > T_{LO} + d\_dn\_wen\_h$ implies that the (timed) path through wire $delay_{WEN}$ is greater than the path through gate NOT_3. In other words, the upper input of latch net_{45} must change before its left input.

Optimization. By replacing within K_1 every parameter except t^{D}_{setup} and t^{WEN}_{setup} by its valuation as defined in π_1, one gets the following constraint on t^{D}_{setup} and t^{WEN}_{setup}:

$$46 < t^{WEN}_{setup} < 54 \ \wedge\ 99 < t^{D}_{setup} \leq 110 \ \wedge\ t^{D}_{setup} < t^{WEN}_{setup} + 61$$

It is then interesting to *minimize* those setup timings. Indeed, if one minimizes the setup duration of the input signals without changing the overall behavior of the system, then this means that the memory can be inserted in a faster environment where the input signals change faster. One can thus minimize t^{D}_{setup} and t^{WEN}_{setup} according to K_1 as follows:

$$t^{WEN}_{setup} = 47 \ \wedge\ t^{D}_{setup} = 100$$

By comparison with the original parameter valuation π_1 (viz., $t^{D}_{setup} = 108$ and $t^{WEN}_{setup} = 48$), this results in a decreasing of the setup timing of signal D (resp. WEN) of 7.4 % (resp. 2.1 %).

In [CEFX06], the authors compute a minimum value of 95 for t^{D}_{setup}, and a minimum value of 29 for t^{WEN}_{setup}. As a consequence, our values may still be improved. Improving those values for this model will be the purpose of Section 5.4.4.

4.7.4 Automatically Generated Model

This second version of the SPSMALL memory is a more complete model of the memory, representing not only the portion of the memory corresponding to the *write* operation, but the complete architecture. As in the previous section, this model was abstracted in order to consider that only one bit is stored. As a consequence, D becomes a 1-bit signal. We give in Figure 4.18 the schematics from [CEFX06] depicting the wires, gates and latches under the form of an Abstract Functional and Timing Graph, and corresponding to the complete architecture of SPSMALL.

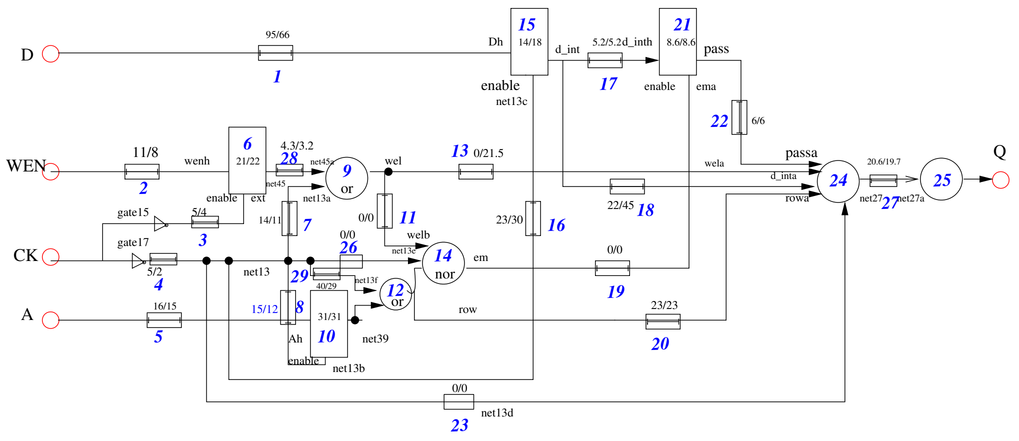


Figure 4.18: Abstract model of the SPSMALL memory

A further major difference with the manual model described in the previous section is that the PTAs are here fully automatically generated. Recall that, in the previous section, the PTAs were written in a partially manual way, and the model was then simplified by grouping together several automata. Here, we first manually wrote the VHDL code corresponding to the different elements of the memory (which is much quicker and less error-prone than describing the PTAs), and then automatically synthesized the PTAs using the tool VHDL2TA [Bar09]. This leads to more parameters, including a slightly richer environment, involving explicitly signal A, characterized by its setup value, viz., t^A_{setup}. This technique results in a model containing 28 automata, 28 clocks, 32 discrete variables and 62 parameters.

Due to the high number of parameters and the complexity of the model, we do not give here the set of parameters valuations coming from the datasheet of SP1 and adapted to this second model, but it can be found in [vWp, And10c]. We only give below the set of parameters valuations (say, π'_1) corresponding to the three input timings we are interested in optimize (timings are given in tens of pico-seconds):

$$t^D_{setup} = 108 \qquad t^{WEN}_{setup} = 48 \qquad t^A_{setup} = 58$$

Applying IMITATOR II to this model and this reference valuation π'_1, one synthesizes a constraint K'_1, projected below onto t^A_{setup}, t^D_{setup} and t^{WEN}_{setup}. The interested reader may refer to [And10c] for the complete valuation and the complete constraint on the whole set of parameters.

$$t^D_{setup} = 108 \ \wedge \ t^{WEN}_{setup} = 48 \ \wedge \ 56 < t^A_{setup} < 60$$

This constraint is an "interesting" (though unfortunate) example of constraint for which the output parameter domain is (almost) reduced to a single point. Thus, it is not possible to optimize values of t_{setup}^{D} and t_{setup}^{WEN} according to this constraint. Nevertheless, the cartography algorithm introduced in Chapter 5 will allow us to overcome this shortcoming, and synthesize a dense set of parameters allowing us to minimize those input timing parameters (see Section 5.4.4).

4.7.5 Larger Models

Two other versions of the SPSMALL memory have been considered. The first one is actually the full SPSMALL memory with 1 memory point of 2 bits. The model described as a network of PTAs has been automatically generated from the VHDL code using VHDL2TA. The VHDL code itself was also automatically generated from the transistor netlist given by ST-Microelectronics. This chain of analysis has been performed in the framework of the VALMEM project (see Figure 4.13 page 99). Unfortunately, because of the high size of this model (101 automata, 101 clocks, 200 parameters, 130 discrete variables, which result in more than 6000 lines of code described in the IMITATOR II syntax), IMITATOR II does not succeed to synthesize a constraint after several hours.

The second version corresponds to a larger version of the SPSMALL memory, with 3 memory points of 2 bits. Due to the even larger size of this model (more than 130 automata), IMITATOR II does not succeed to synthesize a constraint either.

Improving IMITATOR II so that it can synthesize constraints for such large systems is the subject of future work. It is also interesting to note that non-parametric analyses of these two models have been successfully performed using the UPPAAL model checker [LPY97], allowing to verify several properties.

4.8 Networked Automation System

In this section, we consider a Networked Automation System studied in the framework of the SIMOP project of Institut Farman (Fédération de Recherche CNRS, FR3311). This project is a joint work between two laboratories of École Normale Supérieure de Cachan, namely LSV and LURPA. The goal of this project was to define several good behavior zones for a distributed control system, using different techniques of timed verification.

Description of the Model. We are here interested in Networked Automation Systems (NAS). NAS with Ethernet-based fieldbuses (instead of traditional fieldbuses) are more and more often used in the industry, even for critical systems such as chemical or power plants. To ensure the reliability of such systems, not only the functionalities but also the timing performances must be validated.

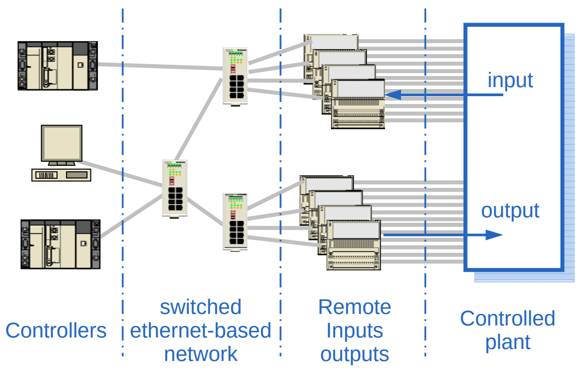


Figure 4.19: Example of Networked Automation Systems (NAS)

The main features of the physical components of these architectures is described in Figure 4.19 (see, e.g., [RDS08, DRF+07]):

- Programmable Logical Controllers (PLCs) are modular. Within each controller, a calculus processor runs a program cyclically, while a communication processor performs a periodic scanning of some Remote Input-Output Modules (RIOMs), termed I/O scanning. It matters to underline that the cycles of these two processors are asynchronous, data exchanges being made by means of a shared memory.

- The network includes Ethernet switches and Ethernet links and is dedicated only to communications between the PLCs and RIOMs; there is no other additional traffic.

- Inputs and outputs from/to the plant are gathered in RIOMs which are directly connected to the network. One RIOM may be shared by several PLCs.

In the following we will use a simple example of NAS, which includes an item of each component: one controller, one Ethernet switch, one RIOM and no particular behavior for the plant. Only one input signal is considered, producing a causal output signal after processing into the controller. Moreover, it will be assumed that there is no frame loss, which is a quite reasonable assumption for this kind of switched industrial Ethernet solution in the concerned operation conditions.

The full description of the model is available in [AAC+09].

In the design and the development process of NAS, engineers have to select and setup components that involves delay parameters. When setting the parameters, the engineer must preserve the expected performance of the NAS, i.e., the response time between the input signal and the output signal. This response time should remain below a maximum limit to get an assessed NAS. The assessment of a NAS is difficult because, for each valuation of the parameters and each input signal, the response time may be different.

The aim of the SIMOP project [AAC$^+$09] is to propose an approach able to assist engineers to design, setup and/or reconfigure Networked Automation Systems, by synthesizing values for the parameters of the NAS guaranteeing a correct response time.

Definition of a Zone of Good Behavior. The system is modeled by a PTA $\mathscr{A}$ containing 7 parameters *COMct*, *COMd*, *NETd*, *PLCct*, *COMct*, *RIOd*, *SIGmrt*, corresponding to various timing delays of the system (see the description of the model in [AAC$^+$09]). We consider the following reference valuation π_0 of the parameters

$$\begin{array}{cccc} PLCct = 600 & COMct = 500 & SIGmrt = 2071 & PLCmtt = 100 \\ RIOd = 70 & COMd = 25 & NETd = 10 & \end{array}$$

It can be shown (e.g., using the model checker UPPAAL [LPY97] for timed automata) that the system under π_0 behaves well (this notion of good behavior corresponds here to the response time of the system being under a given value). The goal of the SIMOP project is to find *other* valuations of the parameters with a good behavior.

Using our program IMITATOR II applied to $\mathscr{A}$ and π_0, we infer the following constraint K_0 defining a good functioning zone.

$$\begin{array}{rl} & 4 * COMct \geq NETd + COMd + 3 * PLCct + PLCmtt \\ \wedge & 4 * COMct \geq RIOd + 2 * NETd + 3 * PLCct + PLCmtt \\ \wedge & PLCct \geq COMct + PLCmtt \\ \wedge & PLCct < RIOd + NETd + COMct + COMd \\ \wedge & SIGmrt > RIOd + 4 * COMct \\ \wedge & NETd > 0 \\ \wedge & PLCmtt > NETd + COMd \\ \wedge & PLCmtt > RIOd + 2 * NETd \\ \wedge & 4 * COMct < RIOd + NETd + COMd + 3 * PLCct + PLCmtt \end{array}$$

Note that this constraint was synthesized using IMITATOR II with the "inclusion" mode. This mode is the implementation of the variant $IM^{\subseteq}$ of IM, which allows to terminate earlier by modifying the fixpoint of the algorithm (see Section 3.5). This good behavior of the NAS is modeled by an observer which checks that the response time is correct, and then goes into a good or a bad state, depending on the response time. As a consequence, although this variant does not guarantee the equality of trace sets, the system under any $\pi \models K_0$ will not contain any bad state because this variant preserves the non-reachability (see Proposition 3.27).

By applying IMITATOR II in the standard mode, the analysis does not terminate, due to the explosion of the state space.

Comparison with Other Methods. In [AAC⁺09], we consider two different approaches: our inverse method, synthesizing a constraint on the parameters, and a dichotomy method, testing (using the UPPAAL model checker) the correctness of a great number of integer points. The dichotomy method synthesizes a cloud of "good" points, which is obviously much bigger than the zone defined by our constraint K_0 (see the graphical comparison in [AAC⁺09]). However, this discrete approach suffers from several limitations. First, only the *discrete* integer points are guaranteed to be correct, whereas our inverse method synthesizes a *dense* zone for which the behavior is guaranteed to be correct. This gives a criterion of robustness for the system, which is interesting in practice, where the real values of the timing delays may not always be exactly equal to the values specified by the designer. Second, only 3 dimensions (viz., *COMct*, *PLCct* and *SIGmrt*) have been considered in the discrete approach, whereas our constraint K_0 is given in 7 dimensions.

The final remarks of [AAC⁺09] suggests the idea to combine both approaches in order to synthesize a much larger dense zone in 7 dimensions: by iterating the inverse method on points synthesized by the dichotomy method, one gets a set of constraints guaranteeing a good behavior. This is actually the idea of the behavioral cartography developed in Chapter 5. However, this idea has not been experimented in the SIMOP project for two reasons. First, the cartography algorithm did not exist at the time of the project. Second, the first version of the tool IMITATOR used in the framework of this project needed almost 7 hours to synthesize a constraint, and iterating it manually would have been highly time-consuming. It would be interesting to investigate this idea again using the new version IMITATOR II, implementing the cartography algorithm.

4.9 Summary of the Experiments

The results of the application of the inverse method to various case studies are given in Table 4.4. We give from left to right the name of the example with its appropriate reference, the number of PTAs composing the global system $\mathscr{A}$, the lower and upper bounds on the number of locations per PTA, the number of clocks and parameters of $\mathscr{A}$, the number of iterations of the algorithm, the number of inequalities within K_0, the number of states and transitions, the computation time in seconds using IMITATOR, and the computation time in seconds using IMITATOR II. Experiments were conducted on an Intel Core2 Duo 2.4 GHz with 2 Gb. Note that the CSMA/CD case study was not detailed in this chapter, but will be mentioned in Section 6.5.2. All case studies are also detailed in [And10c].

Example	PTAs	loc./PTA	$\|X\|$	$\|P\|$	iter.	$\|K_0\|$	states	trans.	Time1	Time2
SR-latch	3	[3,8]	3	3	5	2	4	3	0.11	0.007
Flip-flop [CC07]	5	[4,16]	5	12	9	6	11	10	1.6	0.122
AND–OR [CC05]	3	[4,8]	4	12	14	4	13	13	1.81	0.15
Latch	7	[2,5]	8	13	12	6	18	17	14.4	0.345
CSMA/CD [KNSW07]	3	[3,8]	3	3	19	2	219	342	41	1.01
RCP [KNS03]	5	[6,11]	6	5	20	2	327	518	64	2.3
SPSMALL$_1$ [CEFX09]	10	[3,8]	10	26	32	23	31	30	4680	2.6
BRP [DKRT97]	6	[2,6]	7	6	30	7	429	474	901	34
SIMOP [ACD$^+$09]	5	[5,16]	8	7	53	9	1108	1404	23455	67
SPSMALL$_2$ [CEFX09]	28	[2,11]	28	62	94	45	129	173	-	461

Table 4.4: Summary of experiments for the inverse method

The SPSMALL$_1$ case study corresponds to the model manually written and described in Section 4.7.3. The SPSMALL$_2$ case study corresponds to the model automatically generated and described in Section 4.7.4. Both computation times refer to the first implementation of the memory (SP1, with the reference valuation π_1). It is impossible to analyze the version automatically generated (SPSMALL$_2$) using the first version of IMITATOR because HYTECH runs out of memory when trying to statically compose the 28 automata in parallel.

When considering the cyclicity of the trace sets, note that, for the respective reference valuation considered, the trace sets of the following case studies are acyclic: SR-latch, flip-flop, latch, SPSMALL$_1$, BRP and SPSMALL$_2$. The other trace sets (viz., AND–OR, CSMA/CD, RCP and SIMOP) are cyclic and thus feature infinite behavior.

Note that the computation time using IMITATOR II has dramatically decreased compared to IMITATOR for all examples: the time has been divided at least by 10, and up to 2000 for the SPSMALL$_1$ memory. Explanations for this high improvement are the rewriting of the tool using a library of convex polyhedra instead of the call to HYTECH, the on-the-fly composition of the different PTAs, and the optimization of the algorithm described in Section 4.1.2.

4.10 Tools Related to IMITATOR II

IMITATOR II has been designed to implement the inverse method and the cartography algorithm and, as far as we know, it is the only tool implementing those algorithms. As a consequence, it is not possible to compare directly the features and computation times of IMITATOR II with other tools of the literature. Nevertheless, it is interesting to point out the following tools allowing to perform several kinds of analyses on various classes of timed automata.

One of the first powerful model checkers for analyzing parametric timed automata is HYTECH [HHWT97], which is a tool designed by Henzinger et al. for model checking parametric hybrid systems. HYTECH features an intuitive but powerful input syntax and the capability of performing various computations on hybrid systems, such as (non)reachability analysis, operations on sets of states, etc. Although HYTECH has been used to verify several interesting case studies, it can hardly verify even medium sized examples for two reasons. First, its exact arithmetics with limited precision often leads to overflows. Second, it performs an *a priori* composition of the timed automata given as input, thus preventing the designer from verifying more than a dozen of automata in parallel.

The tool PHAVer [Fre05b] has been designed by Goran Frehse in particular to overcome HYTECH's weaknesses. It highly improves the scalability compared to HYTECH, and performs analyses on parametric hybrid systems using exact arithmetics with unlimited precision and convex polyhedra (using the *Parma Polyhedra Library* (PPL) [BHZ08]). Moreover, PHAVer offers various features such as automatic partitioning, graphical outputs, and forward/backward abstraction refinement. Various case studies have been verified, in particular in the framework of analog circuits [FKR06].

The KRONOS model checker [Yov97] allows to verify real-time systems modeled using networks of timed automata. The properties to be verified are expressed using the real-time temporal logic TCTL [ACD93]. Various case studies in the framework of hardware circuits or communication protocols have been studied [DY95, MY96, TY98].

UPPAAL is a powerful tool for modeling timed systems modeled as networks of timed automata extended with several data types [LPY97]. In particular, it verifies very efficiently timing properties such as reachability, safety or liveness properties on timed automata. Various extensions have been developed for frameworks such as timed games or probabilistic systems. However, although an extension of UPPAAL allowing to perform parametric model checking is mentioned in [ABB$^+$01], the standard version of UPPAAL does not allow the verification of hybrid or parametric systems.

TREX [ABS01] is a model checker allowing to verify properties on parametric timed automata extended with integer counters and finite-domain variables. TREX features on-the-fly verification of safety properties, as well as parameter synthesis either using parametric reachability, or in order to satisfy properties. Various representations are allowed and both forward and backward exploration algorithms can be used. Also note that TREX allows the synthesis of non-linear constraints on the parameters.

The RED library [Wan06] features analysis of real-time systems using Clock-Restriction Diagrams, as well as parametric analysis of hybrid systems using Hybrid-Restriction Diagrams.

The TINA toolbox (TIme petri Net Analyzer) [BRV04, BV06] is a tool allowing the construction of reachability graphs in the framework of Time Petri Nets. It features the computation of the coverability graph of a Petri Net, the marking graph of a bounded Petri net, and allows various state space abstractions for Time Petri nets, possibly preserving some temporal logics (LTL and CTL*) properties. It also features an editor for graphically or textually describing Time Petri Nets.

Finally, the tool Roméo [LRST09] is a software for Time Petri Nets analysis, making use of the UPPAAL DBP library and the Parma Polyhedra Library [BHZ08]. It allows TCTL model-checking for (bounded) Time Petri Nets and for stopwatch models. An interesting feature allows to check *parametric* TCTL formulae, and thus synthesize parameters valuations for which the formula is satisfied. These sets of parameters valuations are expressed using linear constraints allowing the use of disjunctions.

Chapter 5

Behavioral Cartography

– Didn't you say you could read a map? We're miles off!
– So what's the big deal? Turn around and find the road.

Happy Together
(Wong Kar-wai)

In Chapter 3, we introduced the inverse method, allowing to synthesize constraints on a system modeled by parametric timed automata. Starting from a reference valuation of the parameters, the inverse method outputs a constraint such that, for any valuation satisfying this constraint, the trace set of the system is the same as under the reference valuation. The inverse method, and its implementation IMITATOR II described in Chapter 4, allowed us to synthesize constraints guaranteeing the good behavior of various examples of asynchronous circuits and communication protocols. However, the inverse method suffers from two limitations. First, the constraint output by the method is not necessarily maximal, i.e., there may exist parameter valuations outside the constraint such that the behavior is the same as under the reference valuation. Second, the method focuses on the equality of trace sets, which can be seen as a rather strong property, because the good behavior of a timed system can correspond to different trace sets.

In this chapter, we introduce a novel approach for solving the following good parameters problem: Given a PTA $\mathscr{A}$ and a rectangular real-valued parameter domain V_0, what is the largest set of parameters values for which $\mathscr{A}$ behaves well? This approach is based on the inverse method, and overcomes its limitations. Indeed, we show that, by iterating the inverse method of Chapter 3 on the integer points of the rectangular parametric domain V_0, we are able to decompose the parametric space into *behavioral tiles*, i.e., parameter zones with a uniform time-abstract behavior. Then, according to a property on traces that one wants to check, it is easy to partition the parametric space into a subset of "good" tiles (which correspond to "good behaviors") and a subset of "bad" ones. This gives us a *behavioral cartography* of the system.

Recall that $\mathscr{A}$ has a good behavior if it satisfies a certain set of properties invariant for automata having the same set of traces. This is in particular the case of linear-time properties [BK08].

Often in practice, what is covered by the behavioral cartography algorithm is not the *bounded* and *integer* subspace of the parameter rectangle V_0, but two major extensions: first, not only the integer points but all the *real-valued* points of the rectangle are covered by the tiles; second, the tiles are often unbounded and cover most of the parametric space *beyond* V_0. Although the cartography may contain holes, i.e., zones not covered by the algorithm, we give sufficient condition for the full coverage of the real-valued bounded parameter domain.

A major interest is that this behavioral cartography does not depend on the property one wants to verify: only the partition into good and bad tiles actually does. As a consequence, when verifying other properties, it is sufficient to check the property for only one point in each tile in order to get the new partition.

Plan of the Chapter. We first recall in Section 5.1 the good parameters problem, using a motivating example. In Section 5.2, we introduce the behavioral cartography algorithm. We give properties of the algorithm in Section 5.3, and present various case studies in Section 5.4, analyzed using IMITATOR II. We finally present related work in Section 5.5.

5.1 Beyond the Inverse Method

In Chapter 3, we were able to synthesize a set of parameters, under the form of a convex linear constraint, preserving the time-abstract behavior, i.e., the trace sets. However, this method suffers from several drawbacks. First, the synthesized set of parameters is not necessarily maximal, i.e., there may exist other valuations (outside this set) having the same trace set. As pointed out in Proposition 3.13, this maximal set may not exist under a convex form. Moreover, the equality of trace sets may appear too strong a condition: a system designer may want to allow different behaviors (i.e., different trace sets), provided those behaviors are all considered as "good" with respect to some property.

As a consequence, we are interested here in synthesizing a set of parameters (possibly the maximal one) such that, for any parameter valuation, the system behaves well with respect to some property on trace. Formally, we recall below the *good parameters problem* as defined in Section 2.3.

> **The Good Parameters Problem**
> Given a PTA $\mathscr{A}$ and a rectangular real-valued parameter domain V_0, what is the largest set of parameters values within V_0 for which $\mathscr{A}$ behaves well?

In the following, we will aim at solving this problem by iterating on the inverse method defined in Chapter 3.

5.2 The Behavioral Cartography Algorithm

By iterating the inverse method *IM* of Chapter 3 over all the *integer* points of a rectangle[1] V_0 (of which there are a finite number), one is able to decompose (most of) the parametric space included into V_0 into behavioral tiles. We give the behavioral cartography algorithm *BC* in Algorithm 6 [AF10].

Algorithm 6: Behavioral cartography algorithm $BC(\mathscr{A}, V_0)$

```
input  : A PTA $\mathscr{A}$
input  : A finite rectangle $V_0 \subseteq \mathbb{R}_{\geq 0}^M$
output: Tiling: list of tiles (initially empty)
1 repeat
2     select an integer point $\pi \in V_0$;
3     if $\pi$ does not belong to any tile of Tiling then
4         Add $IM(\mathscr{A}, \pi)$ to Tiling;
5 until Tiling contains all the integer points of $V_0$;
```

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

In many cases, all the real-valued space of V_0 is covered by *Tiling* (see case studies in Section 5.4). Besides, the space covered by *Tiling* often largely exceeds the limits of V_0 (see Section 5.3 for a sufficient condition of full coverage of the parametric space).

Partition Between Good and Bad Tiles. If now a decidable trace property is given then one can check which tiles are good (i.e., the tiles whose trace set satisfies the property), and which ones are bad. One can thus partition the rectangle V_0 into a good (resp. bad) subspace, i.e., a union of good (resp. bad) tiles.

[1]Actually, V_0 can be more generally a convex set containing a finite number of integer points.

Advantages. First, the cartography itself does not depend on the property one wants to check. Only the partition between good and bad tiles involves the considered property.

Moreover, the algorithm is interesting because one does not need to compute the set of all the reachable states. On the contrary, each call to the inverse method algorithm quickly reduces the state space by removing the incompatible states. This allows us to overcome the state space explosion problem, which prevents other methods, such as the computation of the whole set of reachable states (and then the intersection with the bad states) [HWT96], to terminate in practice.

Also note that the cartography algorithm makes use of no approximation.

5.3 Properties

In this section, we show that for acyclic PTAs (see Definition 2.24), a variant of the cartography algorithm allows us to cover the whole real-valued space of parameters within V_0.

Lemma 5.1 (Termination). *Given an acyclic PTA $\mathscr{A}$ and a rectangle V_0, the algorithm $BC(\mathscr{A}, V_0)$ always terminates.*

Proof. Based on the termination of the inverse method (see Proposition 3.9) and the finite number of integer points in V_0. ■

Note that, just as for the inverse method, the acyclicity of the PTA is a sufficient, but non-necessary, termination condition of *BC*. We will provide in Section 6.6.2 an example of non acyclic PTA for which the cartography algorithm terminates.

The algorithm *BC* guarantees to cover the *integer* points within V_0. However, there may exist a finite number of "small holes" within V_0 (containing no integer point) that are not covered by any tile of *Tiling*. A possible refinement of the algorithm is to consider a tighter grid, i.e., not only integer points, but rational points multiple of a smaller step than 1.

This algorithm, say BC', is similar as *BC*, except that it takes one more parameter as input, viz., the *step* between two points on which the inverse method shall be called. As a consequence, instead of calling the inverse method on all the *integer* points, it will be called on all rational point multiple of the chosen step.

Acyclic PTAs. In the case of acyclic PTAs, we show in the following that the termination of BC' is guaranteed, and allows to cover the whole parameter space using a step small enough. This is due to the *finiteness* of the number of different tiles which can be output by $IM(\mathscr{A}, \pi)$, for any *rational* point π. Formally:

Lemma 5.2 (Finite number of tiles)**.** *Let $\mathscr{A}$ be an acyclic PTA. The set of tiles $\{IM(\mathscr{A}, \pi) \mid \pi \in \mathbb{Q}^M_{\geq 0}\}$ is finite.*

Proof. First, the number of possible trace sets is finite from the acyclicity of $\mathscr{A}$. Moreover, the set $Post^*_{\mathscr{A}(K)}(\{s_0\})$ of reachable states is finite for any K, due to the acyclicity of $\mathscr{A}$. As a consequence, for a given K, the number of π_0-inequalities is also finite. Thus, there is a finite number of possibilities to refine K by the addition of the negation of a π_0-inequality. As a consequence, there is a finite number of possible constraints K at the end of the algorithm. And, from the finiteness of the number of possible trace sets, the number of possible intersections of K with the constraints associated with the reachable states (i.e., K_0) is finite, which proves the result. ■

Now, one can show that BC' covers the whole parametric space, for a "sufficiently large" V_0 and a step "sufficiently small", for acyclic PTAs. Formally:

Proposition 5.3. *Let $\mathscr{A}$ be an acyclic PTA. Then there exist a rectangle V_0 and a step such that $BC'(\mathscr{A}, V_0)$ covers the whole real-valued parametric space.*

Proof. From the finiteness of the number of tiles possibly output by IM (see Lemma 5.2). ■

General Case. For the general case (i.e., possibly cyclic PTAs), it is also possible to identify classes of systems for which the full coverage of the rectangle V_0 is guaranteed using the classical version BC of the behavioral cartography algorithm. In particular, this is the case of PTAs corresponding to the following requirements:

1. the set P of $2M$ parameters is partitioned into a subset $P^-\{p_1^-, \dots, p_M^-,\}$ and a subset $P^+\{p_1^+, \dots, p_M^+,\}$ of parameters, with $p_i^- \leq p_i^+$, for all $i = 1, \dots, M$;

2. invariants are all of the form $x \leq \sum_{1 \leq i \leq M} \alpha_i p_i^+$, with $\alpha_i \in \mathbb{N}$, for all $i = 1, \dots, M$;

3. guards are all of the form $x \geq \sum_{i \leq i \leq M} \alpha_i p_i^-$, with $\alpha_i \in \mathbb{N}$, for all $i = 1, \dots, M$.

See [Sou10a] for more details and proofs. Although this class of PTAs may seem restrictive, this is actually the case of most of the case studies we met in practice, particularly in hardware verification.

Actually, the finiteness of the number of tiles, and thus the full coverage of V_0, can also be proven in more general cases. For example, the proof of Lemma 5.2 can be adapted to show the finiteness of the number of tiles when the reachability graph of $\mathscr{A}(\texttt{true})$ is finite, i.e., when there exists $n \in \mathbb{N}$ such that $Post^n_{\mathscr{A}(\texttt{true})}(\{s_0\}) \sqsubseteq \bigcup_{j=0}^{n-1} Post^j_{\mathscr{A}(\texttt{true})}(\{s_0\})$.

Remark 5.4. In order to fill the possible holes when using the standard version *BC* of the behavioral cartography algorithm, it may be more efficient in practice to proceed as follows:

1. call the standard version *BC* of the behavioral cartography algorithm (i.e., by calling the inverse method on integer points);
2. fill the possible holes by calling again manually the inverse method on one (non-integer) point within each hole.

This is often more efficient in practice. In the case of acyclic PTAs, the termination of this variant is guaranteed. This is also due to the *finiteness* of the number of different tiles which can be output by $IM(\mathscr{A}, \pi)$, for any *rational* point π of V_0.

Finally note that an interesting variant of the algorithm would be a *dynamic* cartography, where the step would be automatically refined in order to fill the possible holes. This is the subject of ongoing work. □

5.4 Case Studies

We consider a range of case studies, asynchronous circuits and telecommunication protocols, and synthesize constraints for each those case studies. We then compare the constraints synthesized by our method with constraints from the literature, when applicable. We give for each case study sufficient details to understand the model. For a fully detailed description, refer to [And10c].

We first introduce the implementation within IMITATOR II of the behavioral cartography algorithm (Section 5.4.1). We then present a range of case studies, i.e.:

- a “SR-latch” circuit (Section 5.4.2),
- the flip-flop circuit introduced in Chapter 1 (Section 5.4.3), and a variant for another environment, and
- two different abstractions of the SPSMALL memory (Section 5.4.4).

We summarize the experiments in Section 5.4.5.

5.4.1 Implementation

The behavioral cartography algorithm has been implemented in the tool IMITATOR II [And10a], already mentioned in Section 4.1.2 in the framework of the inverse method.

When calling IMITATOR II to apply the behavioral cartography algorithm, the tool takes as input two files, one describing the network of PTAs modeling the system, and the other describing the reference rectangle, i.e., the bounds to consider for each parameter. As depicted in Figure 5.1, it synthesizes a list of tiles, as well as the trace set corresponding to each tile under a graphical form. The description of all the parametric reachable states for each tile is also returned.

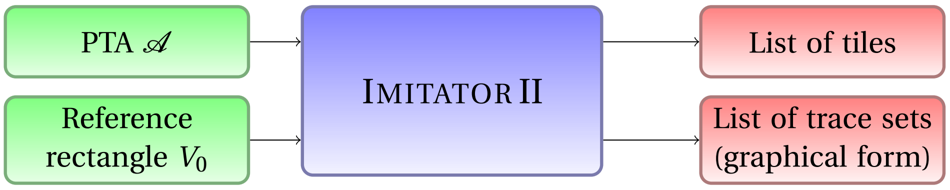

PTA $\mathscr{A}$
Reference rectangle V_0
IMITATOR II
List of tiles
List of trace sets (graphical form)

Figure 5.1: IMITATOR II inputs and outputs in cartography mode

Two different modes can be considered for this algorithm:

1. cover all the integer points of V_0, or
2. call a given number of times the inverse method on an integer point selected randomly within V_0 (which is interesting for rectangles containing a very big number of integer points but few different tiles).

For both modes, the inverse method is not called if the selected point has already been covered by some of the tiles in *Tiling*.

For systems with only two parameter dimensions, the cartography is also automatically returned under a graphical form using the graph utility of gnuplot, a portable command-line driven graphing utility [pWpa]. An example[2] of such an output is given in Figure 5.2, where each tile is depicted with a color. The reference rectangle V_0 is depicted in dashed. The white zone corresponds to points which are not covered by any tile. Note that, due to a limitation of the number of colors (actually only 4) of the external tool allowing to generate automatically the cartography, two tiles depicted with the same color do not necessarily correspond to the same trace sets.

[2]This example actually corresponds to the Root Contention Protocol, which will be studied in Section 6.6.

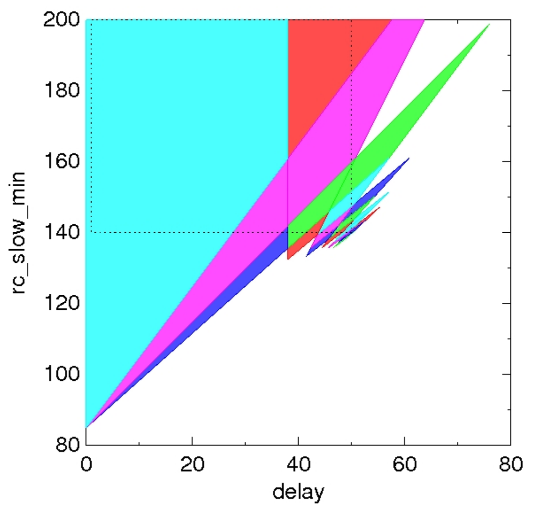


Figure 5.2: Example of cartography automatically output by IMITATOR II

Note that the variant presented in Section 5.3 (i.e., BC') has also been implemented. In other words, the step (by default, the integers) can be given as an input, and the analysis is then performed automatically using this step.

5.4.2 SR-Latch

We consider the SR-latch described in Section 4.2. We now perform a behavioral cartography of this system in order to synthesize a maximal constraint guaranteeing that the system always ends in a state where $\overline{Q} = 1$. We consider the following rectangle V_0 for the parameters.

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

Using IMITATOR II, we get the following six behavioral tiles. For each of those tiles, we will give the corresponding trace set, where the value of the signals corresponding to each location is given in Table 4.1 page 87.

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

$$t^{\downarrow} = \delta_2 \ \wedge \ \delta_1 = 0$$

The trace set of this tile is given in Figure 5.3.

Since $t^{\downarrow} = \delta_2$, $R^{\downarrow}$ and $\overline{Q}^{\uparrow}$ will occur at the same time. Thus, the order of those two events is unspecified, which explains the choice between going to q_2 or q_3.

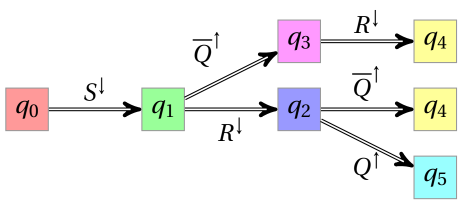

$$\begin{tikzcd}
 & & q_3 \arrow[r, "R^{\downarrow}"] & q_4 \\
q_0 \arrow[r, "S^{\downarrow}"] & q_1 \arrow[ur, "\overline{Q}^{\uparrow}"] \arrow[r, "R^{\downarrow}"'] & q_2 \arrow[r, "\overline{Q}^{\uparrow}"] \arrow[dr, "Q^{\uparrow}"'] & q_4 \\
 & & & q_5
\end{tikzcd}$$

Figure 5.3: Trace set of tile 1 for the SR latch

When in state q_2, either $Q^{\uparrow}$ can occur (since $\delta_1 = 0$), in which case the system is stable, or $\overline{Q}^{\uparrow}$ can occur, which also leads to stability.

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

$$t^{\downarrow} = \delta_2 \ \wedge \ \delta_1 > 0$$

The trace set of this tile is given in Figure 5.4.

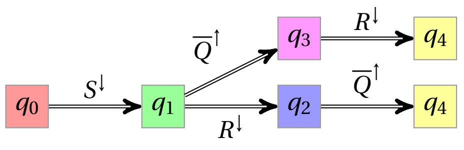

$$\begin{tikzcd}
 & & q_3 \arrow[r, "R^{\downarrow}"] & q_4 \\
q_0 \arrow[r, "S^{\downarrow}"] & q_1 \arrow[ur, "\overline{Q}^{\uparrow}"] \arrow[r, "R^{\downarrow}"'] & q_2 \arrow[r, "\overline{Q}^{\uparrow}"] & q_4
\end{tikzcd}$$

Figure 5.4: Trace set of tile 2 for the SR latch

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

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

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

The trace set of this tile is given in Figure 5.5.

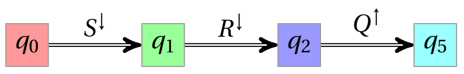

$$\begin{tikzcd}
q_0 \arrow[r, "S^{\downarrow}"] & q_1 \arrow[r, "R^{\downarrow}"] & q_2 \arrow[r, "Q^{\uparrow}"] & q_5
\end{tikzcd}$$

Figure 5.5: Trace set of tile 3 for the SR latch

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

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

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

The trace set of this tile is given in Figure 5.6.

$$\begin{tikzcd}
q_0 \arrow[r, Rightarrow, "S^{\downarrow}"] & q_1 \arrow[r, Rightarrow, "R^{\downarrow}"] & q_2 \arrow[r, Rightarrow, "Q^{\uparrow}"] \arrow[dr, Rightarrow, "\overline{Q}^{\uparrow}"'] & q_5 \\
 & & & q_4
\end{tikzcd}$$

Figure 5.6: Trace set of tile 4 for the SR latch

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

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

$$\delta_2 > t^{\downarrow} \ \wedge \ t^{\downarrow} + \delta_1 > \delta_2$$

Note that this constraint is equal to K_0. The trace set of this tile is given in Figure 5.7.

$$\begin{tikzcd}
q_0 \arrow[r, Rightarrow, "S^{\downarrow}"] & q_1 \arrow[r, Rightarrow, "R^{\downarrow}"] & q_2 \arrow[r, Rightarrow, "\overline{Q}^{\uparrow}"] & q_4
\end{tikzcd}$$

Figure 5.7: Trace set of tile 5 for the SR latch

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

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

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

The trace set of this tile is given in Figure 5.8.

$$\begin{tikzcd}
q_0 \arrow[r, Rightarrow, "S^{\downarrow}"] & q_1 \arrow[r, Rightarrow, "\overline{Q}^{\uparrow}"] & q_3 \arrow[r, Rightarrow, "R^{\downarrow}"] & q_4
\end{tikzcd}$$

Figure 5.8: Trace set of tile 6 for the SR latch

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

Cartography. We give in Figure 5.9 the cartography of this SR-latch case study. For the sake of simplicity of representation, we consider only parameters δ_1 and δ_2. Therefore, we set $t^{\downarrow} = 1$.

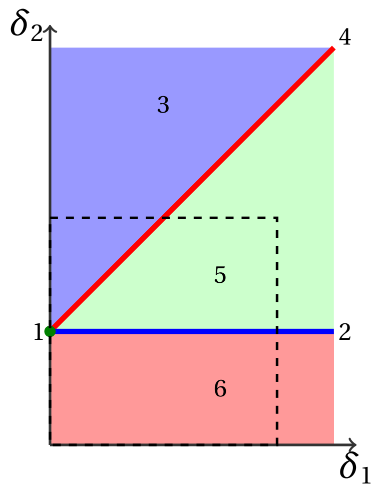


Figure 5.9: Behavioral cartography of the SR latch according to δ_1 and δ_2

The rectangle V_0 is represented with dashed lines. Note that tile 1 corresponds to a point, and tiles 2 and 4 correspond to lines. Note also that all tiles (except tile 1) are unbounded. As a consequence, the cartography covers, not only V_0, but the whole positive real-valued parametric space. Constraints synthesized using our algorithm in order to guarantee a given behavior will thus necessarily be maximal for this case study.

Verification of Properties. Recall that we aim at synthesizing the maximal set of parameters guaranteeing the following behavior: "the system always ends in a state where $\overline{Q} = 1$", i.e., each trace ends either in state q_3 or in state q_4. One can easily infer from the six trace sets that tiles 2, 5 and 6 are good tiles, and the other tiles are bad tiles. As a consequence, the maximal set of parameters corresponding to all the good behaviors is the union of the constraints associated with the three good tiles, i.e.:

$$\begin{array}{ll} & t^{\downarrow} = \delta_2 \ \wedge \ \delta_1 > 0 \\ \vee & \delta_2 > t^{\downarrow} \ \wedge \ t^{\downarrow} + \delta_1 > \delta_2 \\ \vee & t^{\downarrow} > \delta_2 \end{array}$$

It can be shown that this constraint is actually equivalent to $t^{\downarrow} + \delta_1 > \delta_2$. Note that this constraint is *maximal*, because our cartography algorithm covers the whole parametric space.

If one now considers another property, we will get a different partition between good and bad tiles, and thus a different constraint. For example, if one wants to synthesize parameter valuations such that "the system always ends in a state where $\overline{Q} = 0$" (i.e., each trace ends in state q_5), only tile 3 is a good tile, leading to the maximal constraint $\delta_2 > t^{\downarrow} + \delta_1$.

Comparison with other methods. Due to the simplicity of this example, it is possible to apply the method introduced in [HWT96], consisting in computing the whole set of reachable states, and then intersect it with the bad states. We first consider the property "the system always ends in a state where $\overline{Q} = 1$". We introduce one more PTA in parallel with the others, which plays the role of an observer. This PTA goes into a "good" location when synchronizing with action $\overline{Q}^{\uparrow}$, and into a "bad" location when synchronizing with action $Q^{\uparrow}$. Using the HyTech model checker, we can compute the whole set of reachable states, project the constraint onto the parameters, and intersect with "the bad" locations, i.e., keep only the states where the observer is in the bad location. The constraint on the parameters associated with the bad states is $t^{\downarrow} + \delta_1 \leq \delta_2$. Thus, by negating it, one finds back the constraint found by our algorithm, i.e., $t^{\downarrow} + \delta_1 > \delta_2$.

Similarly for the second property (i.e., "the system always ends in a state where $\overline{Q} = 0$"), we slightly modify the observer (i.e., swap the good and the bad location), and this method also allows to synthesize the same constraint as our algorithm.

5.4.3 Flip-flop

We will consider two different versions of the flip-flop case study. The first one is the same as the one described in Section 1.1 and Section 3.1.1. The second one is a variant of this example, using the same model and a different environment.

First environment

We apply here the behavioral cartography to the flip-flop example described in Section 3.1.1. For the sake of simplicity, we consider a model with only 2 parameters, with the following V_0:

$$\delta_3^+ \in [8, 30] \quad \text{and} \quad \delta_4^+ \in [3, 30].$$

The other parameters are instantiated as follows:

$T_{HI} = 24$	$T_{LO} = 15$	$T_{Setup} = 10$	$T_{Hold} = 17$	$\delta_1^- = 7$
$\delta_1^+ = 7$	$\delta_2^- = 5$	$\delta_2^+ = 6$	$\delta_3^- = 8$	$\delta_4^- = 3$

We compute the cartography of the flip-flop circuit according to δ_3^+ and δ_4^+, depicted in Figure 5.10. The dashed rectangle corresponds to V_0.

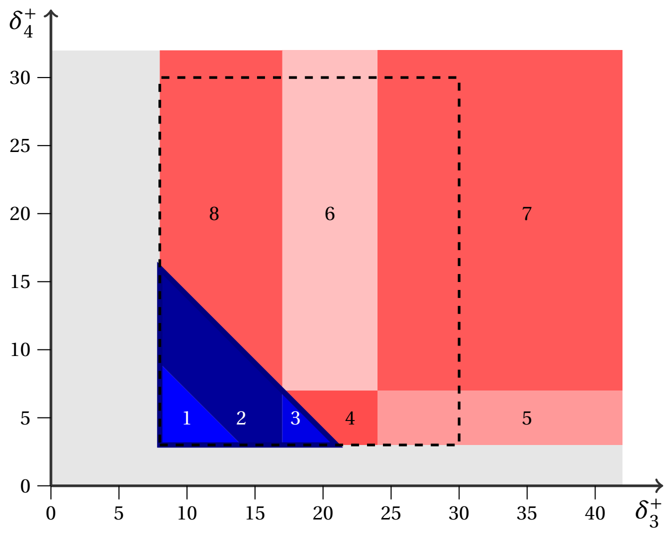


Figure 5.10: Behavioral cartography of the flip-flop according to δ_3^+ and δ_4^+

First note that the whole (real-valued) V_0 is covered. Note also that tiles 5 to 8 are unbounded. Actually, this cartography covers the whole[3] real-valued parametric space $\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$. According to the nature of the trace sets, we can easily partition the tiles into good and bad tiles with respect to property $Prop_1$.

For example, the trace set of tile 3 (corresponding to the constraint $\delta_3^+ + \delta_4^+ < 24 \wedge \delta_3^+ \geq 17 \wedge \delta_4^+ \geq 3$) is given in Figure 5.11, where the meaning of each location in terms of signals is given in Table 3.1 page 43. This tile is a *good* tile because $Q^\uparrow$ occurs before $CK^\downarrow$ for all traces.

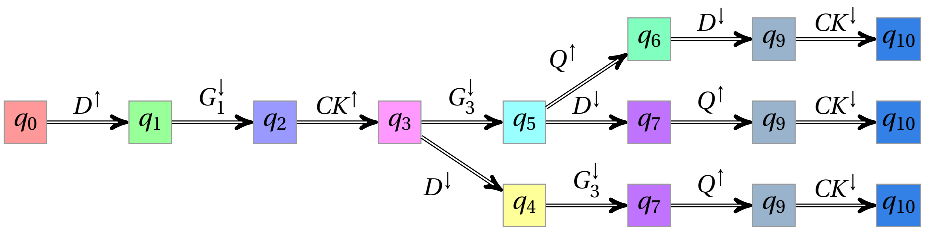

$$\begin{tikzcd}
 & & & & & q_6 \arrow[r, "D^\downarrow"] & q_9 \arrow[r, "CK^\downarrow"] & q_{10} \\
q_0 \arrow[r, "D^\uparrow"] & q_1 \arrow[r, "G_1^\downarrow"] & q_2 \arrow[r, "CK^\uparrow"] & q_3 \arrow[r, "G_3^\downarrow"] \arrow[dr, "D^\downarrow"'] & q_5 \arrow[ur, "Q^\uparrow"] \arrow[r, "D^\downarrow"] & q_7 \arrow[r, "Q^\uparrow"] & q_9 \arrow[r, "CK^\downarrow"] & q_{10} \\
 & & & & q_4 \arrow[r, "G_3^\downarrow"] & q_7 \arrow[r, "Q^\uparrow"] & q_9 \arrow[r, "CK^\downarrow"] & q_{10}
\end{tikzcd}$$

Figure 5.11: Trace set of tile 3 for the flip-flop case study

Likewise, the trace set of tile 7 (corresponding to the constraint $\delta_3^+ \geq 24 \wedge \delta_4^+ \geq 7$) is given in Figure 5.12, where the meaning of each location in terms of

[3]Apart from the irrelevant zone originating from the model ($\delta_3^+ < 8$ or $\delta_4^+ < 3$).

signals is given in Table 3.1 page 43. This is a *bad* tile because there exist traces where $Q^{\uparrow}$ occurs after $CK^{\downarrow}$.

$$\begin{tikzcd}
 & & & & & q_6 \arrow[r, "D^{\downarrow}"] & q_9 \arrow[r, "CK^{\downarrow}"] & q_{10} \\
 & & & & q_5 \arrow[ur, "Q^{\uparrow}"] \arrow[r, "D^{\downarrow}"] & q_7 \arrow[r, "Q^{\uparrow}"] \arrow[dr, "CK^{\downarrow}"] & q_9 \arrow[r, "CK^{\downarrow}"] & q_{10} \\
q_0 \arrow[r, "D^{\uparrow}"] & q_1 \arrow[r, "G_1^{\downarrow}"] & q_2 \arrow[r, "CK^{\uparrow}"] & q_3 \arrow[ur, "G_3^{\downarrow}"] \arrow[r, "D^{\downarrow}"'] & q_4 \arrow[r, "CK^{\downarrow}"] \arrow[dr, "G_3^{\downarrow}"'] & q_{11} & q_8 \arrow[r, "Q^{\uparrow}"] & q_{10} \\
 & & & & & q_7 \arrow[r, "Q^{\uparrow}"] \arrow[dr, "CK^{\downarrow}"'] & q_9 \arrow[r, "CK^{\downarrow}"] & q_{10} \\
 & & & & & & q_8 \arrow[r, "Q^{\uparrow}"] & q_{10}
\end{tikzcd}$$

Figure 5.12: Trace set of tile 7 for the flip-flop case study

One sees more generally that tiles 1 to 3 are good while tiles 4 to 8 are bad. From this partition into good and bad tiles, we infer the following constraint:

$$\delta_3^+ + \delta_4^+ \leq 24 \ \wedge \ \delta_3^+ \geq 8 \ \wedge \ \delta_4^+ \geq 3$$

which gives the *maximal* set of good parameters, thus solving the good parameters problem for this example.

Comparison with other methods. By computing in a brute manner the whole set of reachable states for all possible valuations of the parameters, and performing the intersection with the set of bad locations, we get the same constraint ensuring the good behavior of the system. Note that this comparison is possible because this example is rather simple; for bigger examples, such a computation would be impossible because of the state space explosion problem (see the cartography Root Contention Protocol in Section 6.6.2).

In [CC07], a constraint Z guaranteeing a good behavior is given (see Section 3.4). The projection of this constraint Z onto δ_3^+ and δ_4^+ gives

$$\delta_3^+ < 11 \wedge \delta_3^+ + \delta_4^+ < 18 \wedge \delta_3^+ \geq 8 \wedge \delta_4^+ \geq 3,$$

which is strictly included in our constraint[4].

[4]Actually, the comparison is not completely fair, because the two models are slightly different: in particular, the authors of [CC07] consider an environment where D is initially either equal to 0 or to 1.

Second environment

We now consider a variant of this case study, using the same model, as depicted in Figure 1.1 (left) page 3, the same timing parameters, and the new environment depicted in Figure 5.13.

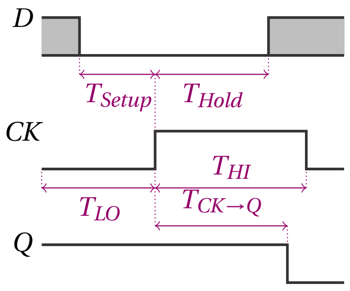


Figure 5.13: Environment for the flip-flop circuit with $D = 0$

This new environment starts from $D = g_2 = Q = 1$ and $CK = g_1 = g_3 = 0$, with the following ordered sequence of actions for inputs D and CK: $D^{\downarrow}$, $CK^{\uparrow}$, $D^{\uparrow}$, $CK^{\downarrow}$. Therefore, we have the implicit constraint $T_{Setup} \leq T_{LO} \wedge T_{Hold} \leq T_{HI}$.

The initial location q_0 corresponds to the initial levels of the signals according to the environment. The initial constraint K_0 corresponds to:

$$T_{Setup} \leq T_{LO} \wedge T_{Hold} \leq T_{HI} \wedge \bigwedge_{i=1,..,4} \delta_i^- \leq \delta_i^+$$

We now consider that the circuit has a *good behavior* if every trace contains both $Q^{\downarrow}$ and $CK^{\downarrow}$, and $Q^{\downarrow}$ occurs before $CK^{\downarrow}$. We are interested in identifying parameter valuations for T_{Hold} and δ_2^+ for which the system has such a good behavior. As a consequence, we perform a behavioral cartography of the system according to parameters T_{Hold} and δ_2^+. We consider the following V_0:

$$T_{Hold} \in [0,50] \quad \text{and} \quad \delta_2^+ \in [5,40].$$

The other parameters are instantiated as follows (note that this reference valuation is not the same as in the previous section):

$$\begin{array}{lllll} T_{HI} = 40 & T_{LO} = 20 & T_{Setup} = 19 & \delta_1^- = 18 & \delta_1^+ = 18 \\ \delta_2^- = 5 & \delta_3^- = 8 & \delta_3^+ = 10 & \delta_4^- = 3 & \delta_4^+ = 7 \end{array}$$

The cartography is computed automatically by IMITATOR II. We then partition the tiles into good and bad. This partition is depicted under a graphical form in Figure 5.14, where the light red (resp. dark blue) zones correspond to the bad (resp. good) values of the parameters.

First note that all outer zones are infinite: as a consequence, the cartography covers the whole[5] dense real-valued set of parameters outside V_0. However,

[5]Apart from the irrelevant zone originating from the model ($\delta_2^+ < 5$).

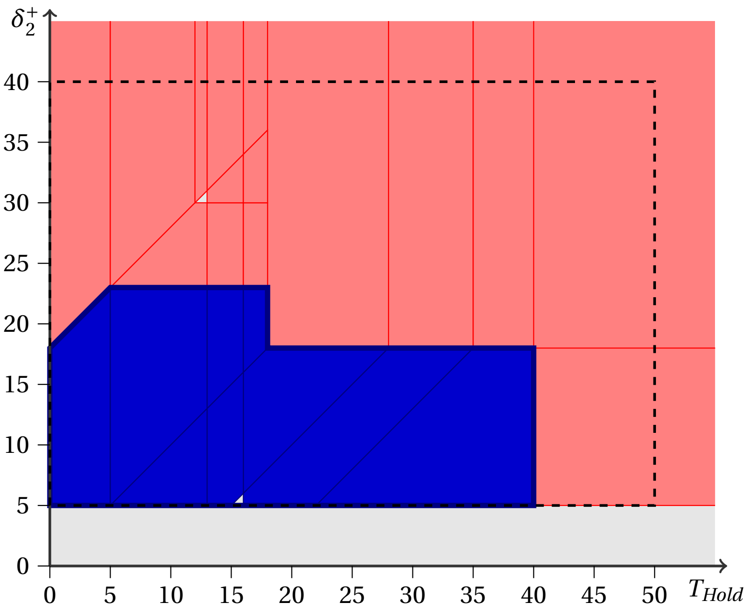


Figure 5.14: Behavioral cartography of the flip-flop for parameters T_{Hold} and δ_2^+

there are two holes within V_0, i.e., zones not covered by any tile. The full coverage can be achieved using two different methods:

1. by calling manually the inverse method on one (non-integer) point within each of the two holes, or

2. by performing again the cartography using a tighter grid than integers (actually calling the inverse method on rational points multiple of 1/3 is enough in this case).

Both methods allow us to get similarly the full coverage of the parametric space within V_0. We do not redraw here the cartography again. The hole in the bad zone turns out to correspond to a bad behavior; similarly, the hole in the good zone turns out to correspond to a good behavior.

As a consequence, one is now able to infer the following constraint corresponding to the set of parameters for which the flip-flop circuit behaves well:

$$\begin{array}{rrcr} & 5 \leq \delta_2^+ \leq 18 & \wedge & 0 \leq T_{Hold} \leq 40 \\ \vee & 18 \leq \delta_2^+ \leq 23 & \wedge & \delta_2^+ - 18 \leq T_{Hold} \leq 18 \end{array}$$

This constraint corresponds to the maximal constraint solving the good parameters problem for parameters T_{Hold} and δ_2^+ for this case study, because the whole parameter domain has been covered by the tiles. Also note that this constraint is not under convex form.

5.4.4 SPSMALL Memory

We consider again the SPSMALL memory, described in Section 4.7. We will consider here two versions of the memory: the manually abstracted model (described and analyzed using the inverse method in Section 4.7.3), and the automatically generated model (described and analyzed in Section 4.7.4).

Manually Abstracted Model

We first consider here the model manually abstracted, described in Section 4.7.3. We are interested in minimizing the values of the setup timing parameters, viz., t^{D}_{setup} and t^{WEN}_{setup}, so that they still verify the following good property mentioned in [CEFX06]: "the response time of the memory must be smaller than 56" (recall that units are given in tens of ps). This response time corresponds to the value $T_{CK \rightarrow Q}$ depicted in Figure 4.15 page 101, and represents the time between the second rise of input signal CK and the rise of the output signal Q. Note that this good property does not strictly speaking correspond to a property on traces. As a consequence, we make use of an *observer* (as in [BC05] and [CEFX06]), i.e., an additional PTA which waits for the rise of Q and, depending on the time of this action, goes into a good location or into a bad location. Locations are observable within traces, thus this property is now a property on traces.

We perform a behavioral cartography of the SPSMALL memory, for the following V_0:

$$t^{D}_{setup} \in [65; 110] \;\; \wedge \;\; t^{WEN}_{setup} \in [0; 66].$$

The other parameters are instantiated like in π_1. We give in Figure 5.15 the cartography of the SPSMALL memory, as automatically output by IMITATOR II. The dashed rectangle corresponds to V_0. The red zone above t^{WEN}_{setup} is infinite, and corresponds to a bad behavior.

Recall that each different colored zone corresponds to a different behavior[6]. Note that the cartography actually contains a few holes, i.e., zones (depicted in white) covered by no tile. We manually "filled" those zones by calling again the inverse method on one point in each zone, which allowed us to cover the whole rectangle V_0.

We then partition the tiles into good and bad. This partition is depicted under a graphical form in Figure 5.16, where the light red (resp. dark blue) zone corresponds to the bad (resp. good) values of the parameters. After partitioning

[6]Recall that this cartography has been automatically output by IMITATOR II which can only represent a few colors (due to the use of an external plot tool). As a consequence, different zones depicted using the same color do not necessarily have the same trace set.

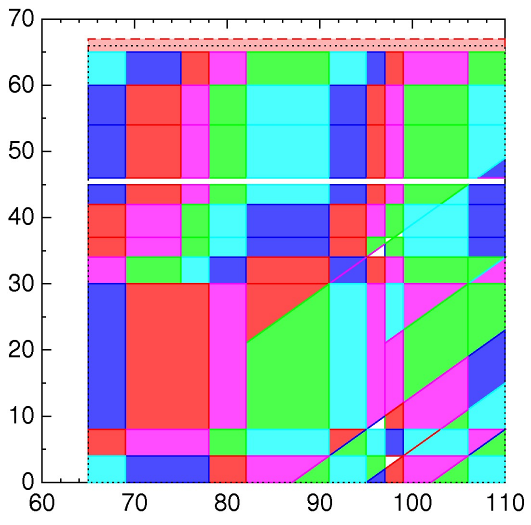


Figure 5.15: Cartography of the SPSMALL memory

the tiles into good and bad, one is able to infer the following constraint corresponding to the set of parameters for which the memory circuit behaves well:

$$99 < t^{D}_{setup} \leq 110 \ \wedge \ 30 < t^{WEN}_{setup} \leq 65$$

This constraint corresponds to the maximal constraint solving the good parameters problem for the SPSMALL memory within V_0, because the whole rectangle has been covered by the tiles.

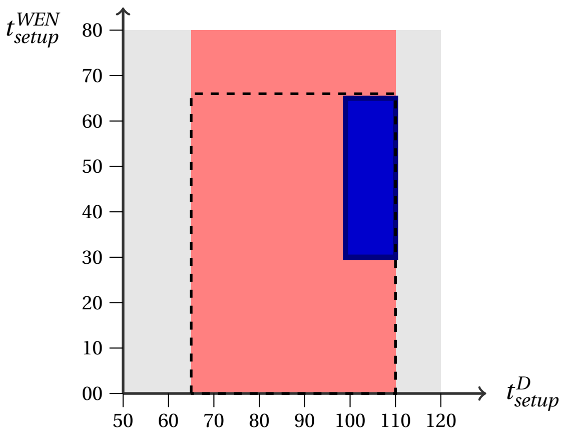


Figure 5.16: Cartography of the SPSMALL memory (after partition)

Due to the way we modeled the system (in particular the environment), values such that $t^{D}_{setup} < 65$ or $t^{D}_{setup} > 110$ do not correspond to any proper behavior. As a consequence, the constraint synthesized corresponds to the maximal constraint for the whole parameter space of this model.

One can thus minimize t^{D}_{setup} and t^{WEN}_{setup} according to the cartography as follows:

$$t^{D}_{setup} = 100 \ \wedge \ t^{WEN}_{setup} = 31$$

By comparison with the original datasheet π_1 (viz., $t^{D}_{setup} = 108$ and $t^{WEN}_{setup} = 48$), this results in a decreasing of the setup timing of signal D of 7.4 %, and a decreasing of the setup timing of signal *WEN* of 35.4 %.

Comparison with Other Methods. In [CEFX06], the authors synthesize a minimum for these setup timings, by iteratively decreasing the setup timings until the system does not behave well anymore, i.e., until the response time is not guaranteed anymore. When compared to our approach, the approach of [CEFX06] has the following limitation: they test only the *integer* points, and do not have any guarantee for the dense set of parameters between two integer points. In [CEFX06], a minimum value of 95 is given for t^{D}_{setup}. However, our approach indicates that the value of 95 corresponds to a bad behavior, and therefore shows a discrepancy between our respective models. A minimum value of 29 is given for t^{WEN}_{setup}, which is slightly smaller as ours. Again, this indicates a discrepancy between our respective models.

Automatically Generated Model

We now consider the model automatically generated, described in Section 4.7.4. As in the previous section, we are interested in minimizing the values of the setup timing parameters, viz., t^{D}_{setup} and t^{WEN}_{setup}, so that they still verify the following good property mentioned in [CEFX06]: "the response time of the memory must be smaller than 56" (recall that units are given in dozens of ps). Again, we make use of an *observer* in order to transform this property into a property on traces.

We perform a behavioral cartography of the SPSMALL memory, for the following V_0:

$$t^{D}_{setup} \in [89; 98] \ \wedge \ t^{WEN}_{setup} \in [25; 34].$$

Due to the complexity of this model, note that the rectangle V_0 is not as large as for the manual model. We give in Figure 5.17 the cartography of the SPSMALL memory, as automatically output by IMITATOR II. The dashed rectangle corresponds to V_0.

Recall that each different colored zone corresponds to a different behavior. This cartography, though interesting, contains many holes, i.e., zones (depicted in white) covered by no tile.

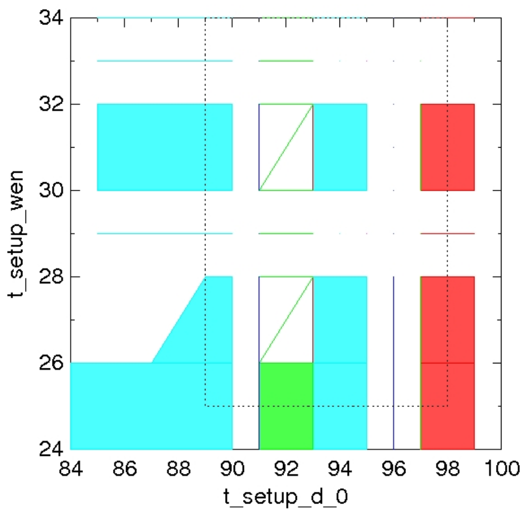


Figure 5.17: Cartography of the SPSMALL memory (generated model)

We then chose to launch again the analysis using a tighter grid, viz., by calling the inverse method on points multiple of 1/3 instead of integer points. This corresponds to the algorithm BC' sketched in Section 5.3. The reason for the choice of 1/3 is that, with such a step, one is sure to cover any tile delimited by integer points. This is not the case of a step of 1 (or even 1/2), because tiles delimited by integer points may *exclude* those integer points in the case of strict inequalities.

This second cartography of the SPSMALL, with step 1/3, is given in Figure 5.18. This cartography is this time successful in the sense that the whole bounded parameter domain V_0 is covered by the tiles. Furthermore, a significant part of the parametric space outside V_0 is also covered.

We then partition the tiles into good and bad. This partition is depicted under a graphical form in Figure 5.19, where the light red (resp. dark blue) zone corresponds to the bad (resp. good) values of the parameters. From this partition, one is able to infer the following constraint corresponding to the set of parameters within V_0 for which the memory circuit behaves well:

$$96 \leq t^{D}_{setup} \leq 98 \;\; \wedge \;\; 29 \leq t^{WEN}_{setup} \leq 34$$

This constraint corresponds to the maximal constraint solving the good parameters problem for the SPSMALL memory within V_0, because the whole rectangle has been covered by the tiles. Also note that the cartography gives further information outside V_0.

One can thus minimize t^{D}_{setup} and t^{WEN}_{setup} according to the cartography as fol-

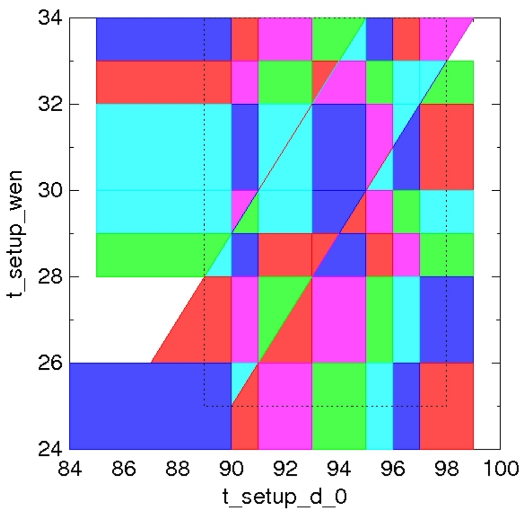


Figure 5.18: Cartography of the SPSMALL memory (full coverage)

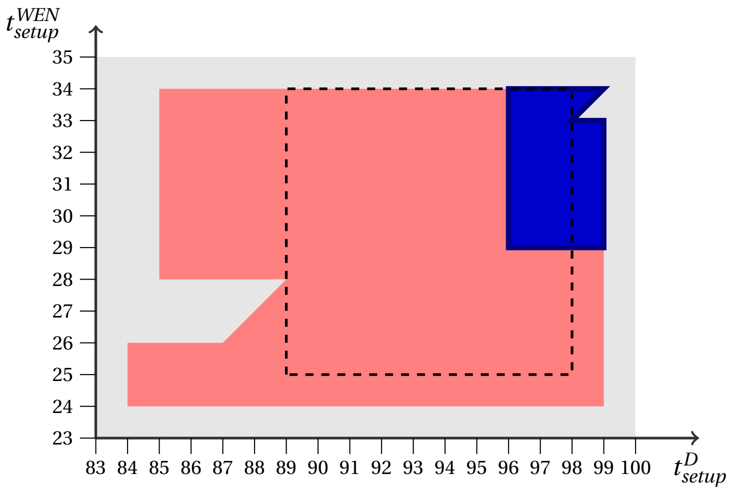


Figure 5.19: Cartography of the generated model of the SPSMALL memory (after partition)

lows:

$$t^{D}_{setup} = 96 \ \wedge \ t^{WEN}_{setup} = 29$$

By comparison with the original valuation for t^{D}_{setup} and t^{WEN}_{setup} (viz., $t^{D}_{setup} = 108$ and $t^{WEN}_{setup} = 48$), this results in a decreasing of the setup timing of signal *D* of 11.1 %, and a decreasing of the setup timing of signal *WEN* of 39.6 %. Such an important decreasing of some of the values of the environment show the interest of the cartography algorithm for the optimization of timing parameters.

Comparison with Other Methods. Recall that, in [CEFX06], the authors also synthesize a minimum for these setup timings, by iteratively decreasing the setup timings until the system does not behave well anymore. In [CEFX06], a minimum value of 95 is given for t^{D}_{setup}. However, our approach indicates that the value of 95 corresponds to a bad behavior, and therefore shows a slight discrepancy between our respective models. Also observe that the authors of [CEFX06] find a minimum value of 29 for t^{WEN}_{setup}, which is exactly the same as ours. This shows the interest of our method, which computes a constraint allowing to retrieve fully automatically the (manually computed) results from [CEFX06], with the advantages that we considered the full model of the memory (not only the write operation), that we give relations between the parameters (under the form of a constraint), and above all that we now give conditions of correctness on the *dense* space of parameters.

Due to the high size of this model (viz., an NPTA composed of 28 PTA containing 28 clocks, 32 discrete variables and 62 parameters) and to the practical interest of the constraint output, this case study can be considered as an extremely interesting application of IMITATOR II.

Remark 5.5. In [CEFX06], values corresponding to simulation are given. Simulation is a technique based on an exact virtual version of the memory. It is usually extremely costly to perform (and is suitable for only one environment) but its results can be considered as exact for this particular case. For this case study, a simulation has been performed using the entire system (i.e., without cutting away some parts of the memory), for some (punctual) values of the input timings. For this environment and those values of the parameters, according to [CEFX06], the minimum possible value computed by simulation for t^{WEN}_{setup} is 36, and the minimum possible value for t^{D}_{setup} is 95. For t^{D}_{setup}, this means that the value we compute is suitable, because it is greater than the minimum possible value. Moreover, it is almost the optimal value, since our method allows to minimize t^{D}_{setup} to 96, whereas the minimum value is 95. For t^{WEN}_{setup}, however, our value is strictly smaller than the value computed using the simulation, which represents a minimum. This indicates that (at least) one delay assigned to a gate of our model (which has been automatically computed in the framework of the VALMEM project) is too approximative. Note that this limitation is of course not due to the methods developed here, but to the way the PTAs and the reference valuation were automatically generated, which is completely beyond the scope of this thesis. □

5.4.5 Summary of the Experiments

The results of the application of the behavioral cartography algorithm to various case studies are given in Table 5.1. We give from left to right the name of the example with its appropriate reference, the number of PTAs composing the global system $\mathscr{A}$, the lower and upper bounds on the number of locations per PTA, the number of clocks of $\mathscr{A}$, the number of parameters varying in the cartography, the number of integer points within V_0, the number of tiles computed, the average number per tile of states and transitions of the trace set, and the computation time in seconds. Experiments were conducted on an Intel Core2 Duo 2.4 GHz with 2 Gb. Case studies not mentioned in this chapter are detailed in [And10c].

Example	PTAs	loc./PTA	$\|X\|$	$\|P\|$	$\|V_0\|$	tiles	states	trans.	Time
SR-latch	3	[3,8]	3	3	1331	6	5	4	0.3
Flip-flop [CC07]	5	[4,16]	5	2	644	8	15	14	3
Latch circuit	7	[2,5]	8	4	73062	5	21	20	96.3
AND–OR [CC05]	3	[4,8]	4	6	75600	4	64	72	118
CSMA/CD [KNSW07]	3	[3,8]	3	3	2000	140	349	545	269
SPSMALL$_1$ [CEFX09]	10	[3,8]	10	2	3149	259	60	61	1194
RCP [KNS03]	5	[6,11]	6	3	186050	19	5688	9312	7018
SPSMALL$_2$ [CEFX09]	28	[2,11]	28	3	784	213	145	196	31641

Table 5.1: Summary of experiments for the cartography algorithm

Recall that the SPSMALL$_1$ case study corresponds to the model manually written, and the SPSMALL$_2$ case study corresponds to the model automatically generated. For the SPSMALL$_2$ case study, the statistics given correspond to the cartography performed using a step of 1/3 and allowing us to cover the whole parameter space within V_0: as a consequence, the number of points within V_0 correspond for this case study, not to the number of integer points, but of rational points multiple of 1/3.

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

Note that, in contrast to the inverse method (Section 4.9), no comparison with the computation time of the first version of IMITATOR is possible, because this first version did not feature the behavioral cartography algorithm.

5.5 Related Work

For work related on parameter synthesis in general in the framework of parametric timed automata, see Section 3.6.

When coming to the good parameters problem with respect to a property one wants to check, the authors of [KP10] (see Section 3.6) synthesize a set of parameter valuations satisfying a given property. This is an interesting way to solve the good parameters problem, in the case where the “good” behavior of the system can be expressed under the form of an ECTL$_{-X}$ formula. In particular, recall that our algorithm does not guarantee the *branching* structure of properties (and thus does not preserve CTL and its variants). However, our cartography algorithm synthesizes sets of valuations both satisfying and not satisfying the formula one is interested in. Indeed, recall that the cartography does *not* depend on the property one considers; only the partition into good and bad tiles does.

One can also see as a kind of behavioral cartography the dichotomy method considered in [AAC$^+$09], and discussed in Section 4.8. Indeed, this method synthesizes a cloud of “good” and “bad” points with respect to a property. However, this discrete approach suffers from several limitations. First, only the *discrete* integer points are guaranteed to be correct (whereas our inverse method synthesizes a *dense* zone for which the behavior is guaranteed to be correct). Second, only a limited number of dimensions can be considered. Third, this dichotomy approach intrinsically depends on the property one wants to verify, whereas our cartography algorithm is independent from the property. As a consequence, for another property, the dichotomy approach should start everything again from the beginning, whereas our algorithm only needs to partition again the computed zones into good and bad by testing only one point in each zone.

To our knowledge, no other method allows a pavement of the dense parametric space with respect to the time-abstract behavior of the system.

Chapter 6

Extension to Probabilistic Systems

Il y a 90 chances pour 100 que je me trompe — mais ça n'a aucune importance.

Ma nuit chez Maud
(Éric Rohmer)

In Chapter 3, we presented the inverse method synthesizing a constraint on the parameters guaranteeing the same time-abstract behavior as under a reference valuation. In Chapter 5, we extended this method in order to synthesize a behavioral cartography of the system. In this chapter, we now extend those results to the probabilistic case.

We consider here the model of *probabilistic timed automata*, where discrete actions in timed automata are replaced with *distributions* of actions. In other terms, in a given location, for a given action, one can reach several locations with distinct probabilities. This formalism is interesting to model probabilistic systems, e.g., randomized protocols.

As for standard timed automata, the behavior of probabilistic timed automata is very sensitive to the values of the constants compared with the clocks within the guards and invariants. It is thus interesting to consider them to be unknown constants, or *parameters*. Verification of probabilistic timed automata models is generally performed with regard to a single reference valuation π_0 of those timing parameters. Given such a parameter valuation, we present in this chapter a method for obtaining automatically a constraint K_0 on timing parameters for which the reachability probabilities (1) remain invariant, and (2) are equal to the reachability probabilities for the reference valuation.

The method relies on parametric analysis of a non-probabilistic version of the probabilistic timed automata model, using the inverse method described in Chapter 3.

This extension of our method to the probabilistic case presents the following advantages. First, as for the non-probabilistic framework, since K_0 corresponds to a dense domain around π_0 on which the system behaves uniformly, it gives us a measure of robustness of the system. Second, it allows us to *rescale* the timing constants used in the system to a valuation in K_0 much smaller than π_0, thus making the probabilistic analysis of the system easier and faster in practice. Indeed, probabilistic analyses of timed systems are often performed using an integer time semantics, which allows to take into account the branching structure of the system. Such analyses using an integer semantics often highly depend on the size of the timing constants used in the system. Therefore giving a formal justification for the rescaling of such constants is of interest for improving the performance of the analysis. Our approach is also useful for avoiding repeated executions of probabilistic model checking analyses for the same model with different parameter valuations.

We also show that the behavioral cartography introduced in Chapter 5 can be applied to probabilistic systems as well, thus allowing to synthesize a "probabilistic cartography" of a system. As a consequence, the value of the reachability probabilities is uniform in each tile.

We provide examples of the application of our technique to models of randomized protocols, and show that the computation time of reachability probabilities drastically decrease when applied to much smaller values than π_0 within K_0.

Plan of the Chapter. We first introduce in Section 6.1 a motivating example, which allows us to informally describe the problem we are interested in solving in this chapter. We then recall in Section 6.2 the framework of probabilistic timed automata, and introduce a syntactic extension to the parametric case, viz., parametric probabilistic timed automata. We then formally state in Section 6.3 the problem we aim at solving in this chapter, using a motivating example of randomized protocol. We show in Section 6.4 how the inverse method of Chapter 3 can be used to compute constraints preserving probabilities of reachability properties in this probabilistic framework. We present in Section 6.5 various case studies of randomized protocols, and give the constraints synthesized using IMITATOR II. In Section 6.6, we then show that the cartography algorithm of Chapter 5 can be applied to the probabilistic framework in a straightforward manner, and apply it to the example of randomized protocol. We finally discuss in Section 6.7 works related to our approach.

6.1 A Motivating Example

We consider here again the Root Contention Protocol, which was mentioned in Section 4.4 in the non-probabilistic framework. Recall that this protocol, used for the election of a leader in the physical layer of the IEEE 1394 standard, consists in first drawing a random number (0 or 1), then waiting for some time according to the result drawn, followed by the sending of a message to the contending neighbor. This is repeated by both nodes until one of them receives a message before sending one, at which point the root is appointed.

The main difference with the model of Section 4.4 is that whereas the draw of the random number was performed in a non-deterministic manner in Section 4.4, we consider it here as a *probabilistic* choice. The timed automaton model given in Section 4.4 now becomes a *probabilistic timed automaton* model. Without going into details (which will be given in further sections), it is possible to compute for such probabilistic timed automata the minimum or maximum probabilities to reach a given location.

Let us consider the minimum probability that a leader is elected after 3 rounds or less. When trying to compute probabilities in the framework of probabilistic timed automata, one can make use of the PRISM model checker [HKNP06]. This model checker actually uses an integer semantics, and uses only integer values for the time stamps and the constants used in the model [KNPS06]. Each elapsing of a time unit is modeled by a transition within a (probabilistic) timed automaton. This integer semantics is of practical interest in order to take into account the branching structure of the system. As a consequence, PRISM is very sensitive to the size of the constants of the model, because big constants quickly lead to the explosion of the state-space.

Recall that the IEEE reference valuation for the 5 parameters of the Root Contention Protocol is the following one:

$$\begin{array}{lll} f\_min = 760 & f\_max = 850 & delay = 360 \\ s\_min = 1590 & s\_max = 1670 & \end{array}$$

In the case of the Root Contention Protocol with those 5 parameters, the PRISM model checker does not succeed in computing this minimum probability that a leader is elected after 3 rounds or less. As a consequence, one usually rescales down the constants. But, because of the integer semantics, one generally needs to round them to the next integer value. This is generally done in a conservative way, as follows: constants appearing as an upper bound in an inequality are rounded up, and constants appearing as a lower bound are rounded down. As a consequence, computed maximum probabilities on the verified probabilistic timed automata model can be greater than those on the hypothetical probabilistic timed automata model. Vice versa, computed minimum probabilities on the verified probabilistic timed automata model can be less than those on the hypothetical probabilistic timed automata model.

Our aim is to synthesize a constraint on the timing constants seen as *parameters* such that, for any valuation of the parameters satisfying this constraint, the minimum and maximum probabilities of reaching a given location is the same. This will allow us to safely rescale the constants without changing the probabilities, and therefore provide us with a formal justification of rescaling and rounding.

6.2 Probabilistic Timed Automata

6.2.1 Timed Probabilistic Systems

A (discrete) probability *distribution* over a countable set Z is a function $\mu : Z \to [0,1]$ such that $\sum_{z \in Z} \mu(z) = 1$. We define $\mathsf{support}(\mu) = \{z \in Z \mid \mu(z) > 0\}$. Then for an uncountable set Z we define $\mathsf{Dist}(Z)$ to be the set of functions $\mu : Z \to [0,1]$, such that $\mathsf{support}(\mu)$ is a countable set and μ restricted to $\mathsf{support}(\mu)$ is a (discrete) probability distribution. A *point distribution* is a distribution $\mu \in \mathsf{Dist}(Z)$ such that $\mu(z) = 1$ for some (unique) $z \in Z$. Often we write μ_z for the point distribution such that $\mu(z) = 1$.

Definition 6.1 (TPS)**.** A *timed probabilistic system (TPS)* is a tuple $\mathcal{T} = (S, S_0, \Sigma, \Rightarrow)$ where: S is a set of *states*, including a set S_0 of *initial states*; Σ is a finite set of *actions* (disjoint from $\mathbb{R}$); $\Rightarrow \subseteq S \times \mathbb{R}_{\geq 0} \times \Sigma \times \mathsf{Dist}(S)$ is a *probabilistic transition relation*. We assume that the probabilistic transition relation is *total*; that is, for every state $s \in S$, there exists $(s, d, a, \mu) \in \Rightarrow$ for some $d \in \mathbb{R}_{\geq 0}, a \in \Sigma, \mu \in \mathsf{Dist}(S)$.

A transition $s \xrightarrow{d,a,\mu} s'$ is made from a state $s \in S$ by first nondeterministically selecting a duration-action-distribution triple (d, a, μ) such that $(s, d, a, \mu) \in \Rightarrow$, and second by making a probabilistic choice of target state s' according to distribution μ, such that $\mu(s') > 0$.

A *path* of a TPS is a non-empty finite sequence of transitions $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} \cdots \xrightarrow{d_{n-1},a_{n-1},\mu_{n-1}} s_n$. Given a path $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} \cdots \xrightarrow{d_{n-1},a_{n-1},\mu_{n-1}} s_n$, we let $last(\omega) = s_n$. The set of paths of a TPS $\mathcal{T}$ is denoted by $Path^{\mathcal{T}}_{fin}$. When clear from the context we omit the superscript $\mathcal{T}$ and write $Path_{fin}$. We let $Path_{fin}(s)$ denote the set of paths commencing in the state $s \in S$. ■

A *scheduler* is a function which chooses an outgoing transition in the last state of a path. Formally, a scheduler of a TPS is a function σ such that, for each path ω of the TPS, if $\sigma(\omega) = (d, a, \mu)$ then $(last(\omega), d, a, \mu) \in \Rightarrow$.

A scheduler resolves the nondeterminism by choosing a transition based on the path executed so far. Intuitively, if a TPS is guided by scheduler σ and has the path ω as its history, then it will be in state s in the next step with probability $\mu(s)$, where $\sigma(\omega) = (d, a, \mu)$. We denote the set of paths induced by a given scheduler σ to be $Path^{\sigma}_{fin} = \{\omega = s_0 \xrightarrow{d_0, a_0, \mu_0} \cdots \xrightarrow{d_{n-1}, a_{n-1}, \mu_{n-1}} s_n \mid \sigma(\omega{\downarrow}_i) = (d_i, a_i, \mu_i)\ \textit{for all}\ i < n\}$, where $\omega{\downarrow}_i$ returns the prefix of ω up to length i. Then we define $Path^{\sigma}_{fin}(s) = Path^{\sigma}_{fin} \cap Path_{fin}(s)$. For each $s \in S$ and scheduler σ, we can define the probability measure $Prob^{\sigma}_{s}$ over measurable sets of paths in the standard way [KSK76].

6.2.2 Probabilistic Timed Automata

Probabilistic timed automata [GJ95, KNSS02] are an extension of classical timed automata [AD94] with discrete probability distributions, and can be used to model probabilistic real-time systems, such as timed randomized protocols or fault-tolerant systems. This probabilistic extension adds discrete probability distributions over edges, so that the choice of the next location of the automaton is not only nondeterministic, but now also probabilistic.

Syntax

We recall below the definition of probabilistic timed automata.

Definition 6.2 (Probabilistic Timed Automaton)**.** A *probabilistic timed automaton* $\mathscr{A}$ is a tuple $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$, where:

- Σ is a finite set of *actions*,
- Q is a finite set of *locations* with an *initial location* $q_0 \in Q$,
- X is a set of *clocks*,
- I is the *invariant* function, assigning to every $q \in Q$ a constraint $I(q)$ on the clocks X, and
- $\rightarrow$ is the *probabilistic edge relation* consisting of elements of the form (q, g, a, η), where $q \in Q$, g is a constraint on the clocks X, $a \in \Sigma$, and $\eta \in \mathsf{Dist}(2^X \times Q)$.

■

We follow the following conventions for the graphical representation of probabilistic timed automata: locations are represented by nodes, above of which the invariant of the location is written; probabilistic edges are represented by arcs from locations, labeled by the associated guard and action, and which split into multiple arcs, each of which leads to a location and is labeled by a set of clocks and a probability (probabilistic edges which correspond to probability 1 are illustrated by a single arc from location to location). Note that, like for TAs, we omit guards and invariants equal to `true`.

Example 6.3. We give in Figure 6.1 an example of probabilistic timed automaton, containing 4 locations (viz., q_0, q_1, q_2 and q_3) and 2 clocks (viz., x and y).

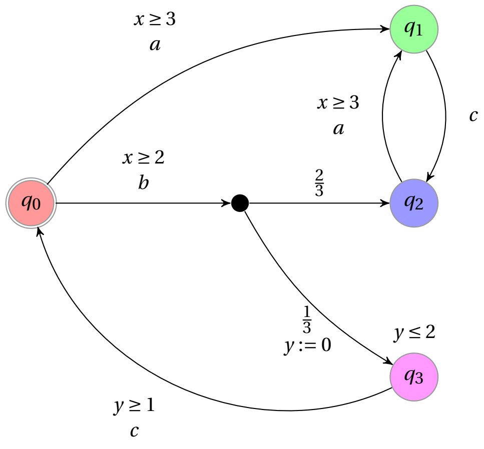


Figure 6.1: Example of probabilistic timed automaton

From location q_0, one can choose nondeterministically between actions a and b. When choosing action a, one can reach location q_1 (with probability 1) if the guard $x \geq 3$ is satisfied. When choosing action b, if the guard $x \geq 2$ is satisfied, then one can reach location q_2 with probability $2/3$, or reach location q_3 with probability $1/3$ and reset y. The rest of this probabilistic timed automaton does not feature probabilistic edges, and can be explained in a similar way as for timed automata. □

Assumptions. We make the following syntactic assumptions on probabilistic timed automata.

Determinism on actions: Given a location $q \in Q$ and action $a \in \Sigma$, there is at most one probabilistic edge of the form $(q, \_, a, \_) \in \rightarrow$.

Reset unicity: For any probabilistic edge $(q, g, a, \eta) \in \rightarrow$ and location $q' \in Q$, there exists at most one $\rho \in 2^X$ such that $\eta(\rho, q') > 0$.

Example 6.4. In Figure 6.2 we give an example of a probabilistic timed automaton fragment which satisfies neither determinism on actions nor reset unicity (left), and an example of a probabilistic timed automaton fragment which satisfies both assumptions (right).

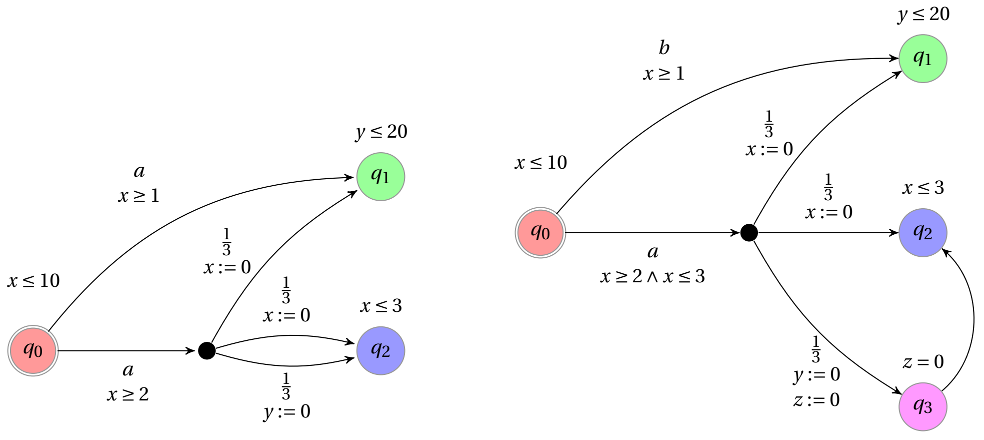


Figure 6.2: Examples of probabilistic timed automata satisfying neither determinism on actions nor reset unicity (left) and satisfying both (right)

The probabilistic timed automaton fragment on the left does not satisfy determinism on actions, because there are two probabilistic edges labeled by a exiting the location q_0; it also does not satisfy the assumption of reset unicity, because the lower probabilistic edge has two distinct probabilistic alternatives which lead to location q_2.

□

Neither determinism on actions nor reset unicity is restrictive, because a probabilistic timed automaton not satisfying the assumptions can be transformed into a probabilistic timed automaton which does: for determinism on actions, it is necessary to add and rename actions, whereas, for reset unicity, it suffices to add an extra clock and additional locations (for example, see the use of the clock z and location q_3 in the right-hand probabilistic timed automaton in Figure 6.2). The assumptions of determinism on actions and reset unicity are commonly met in practice, and they simplify the proofs of our subsequent results.

Note that the probabilistic timed automaton considered in Example 6.3 satisfies both assumptions.

Network of Probabilistic Timed Automata. Networks of Probabilistic Timed Automata can be defined by using parallel composition based on the synchronization of discrete transitions of different components sharing the same action in a similar manner to networks of TAs. For the sake of simplicity, we will suppose that synchronized actions are non-probabilistic (their distribution is a point distribution).

The following definition is similar to the notion of parallel composition of Probabilistic Timed Automata of [KNS03], with the difference that we here consider N probabilistic timed automata (instead of 2 in [KNS03]).

Definition 6.5 (Network of Probabilistic Timed Automata)**.** Let $N \in \mathbb{N}$. For all $1 \leq i \leq N$, let $\mathscr{A}_i = (\Sigma_i, Q_i, (q_0)_i, X_i, I_i, \rightarrow_i)$ be a probabilistic timed automaton. The sets Q_i and X_i are mutually disjoint. A *network of probabilistic timed automata* is $\mathscr{A} = \mathscr{A}_1 \| \ldots \| \mathscr{A}_N$, where $\|$ is the operator for parallel composition defined in the following way. This network of probabilistic timed automata corresponds to the probabilistic timed automaton $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$ where

- $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,
- $Q = \Pi_{i=1}^{N} Q_i$,
- $q_0 = \langle (q_0)_1, \ldots, (q_0)_N \rangle$,
- $X = \biguplus_{i=1}^{N} X_i$,
- $I(\langle q_1, \ldots, q_N \rangle) = \bigwedge_{i=1}^{N} I_i(q_i)$ for all $\langle q_1, \ldots, q_N \rangle \in Q$,

and $\rightarrow$ is defined as follows.

We first define the notation $\otimes$. Given N distributions $\eta_i \in \mathsf{Dist}(2^{X_i} \times Q_i)$, with $1 \leq i \leq N$, we define $\eta_1 \otimes \cdots \otimes \eta_N \in \mathsf{Dist}(2^X \times Q)$ in the following way: for each $\rho_1 \in X_1, \ldots, \rho_N \in X_N$ and $q = \langle q_1, \ldots, q_N \rangle \in Q$, let $\eta_1 \otimes \cdots \otimes \eta_N(\biguplus_{i=1}^{N} \rho_i, q) = \prod_{i=1}^{N} \eta_i(\rho_i, q_i)$.

For all $a \in \Sigma$, let T_a be the subset of indices $i \in 1, \ldots, N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $\langle q_1, \ldots, q_N \rangle \in Q$, for all $\langle q'_1, \ldots, q'_N \rangle \in Q$, we have that $(\langle q_1, \ldots, q_N \rangle, g, a, \eta) \in \rightarrow$ if:
for all $i \in T_a$, there exist $(q_i, g_i, a, \eta_i) \in \rightarrow_i$ such that

- $g = \bigwedge_{i \in T_a} g_i$, and
- $\eta = \bigotimes_{i \in T_a} \eta_i \otimes \bigotimes_{i \notin T_a} \mu_{(\emptyset, q_i)}$.

■

Semantics

In this section, we will consider the probabilistic timed automaton $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$.

A *state* of $\mathscr{A}$ is a pair $(q, w) \in Q \times \mathbb{R}^H_{\geq 0}$ such that $w \models I'(q)$. Informally, the behavior of $\mathscr{A}$ can be understood as follows. The model starts in the initial location q_0 with all clocks set to 0. In this, and any other state (q, w), there is a nondeterministic choice of (1) the amount of time which then passes, and (2) which discrete transition is then taken. Note that, for point (1), time can pass only if invariant $I'(q)$ is satisfied while time elapses. Furthermore, for point (2), a discrete transition can be made according to any probabilistic edge $(q, g, a, \eta) \in \rightarrow'$ with source location q which is *enabled*; that is the constraint g is satisfied by the current clock valuation w. Then the probability of moving to the location q' and resetting all of the clocks in ρ to 0 is given by $\eta(\rho, q')$.

A probabilistic timed automaton can be interpreted as an infinite TPS. Due to the continuous nature of clocks, the underlying TPS has uncountably many states, and are uncountably branching. A probabilistic timed automaton can thus be considered as a *finite* description of infinite TPS. Formally, we define as follows the semantics of a probabilistic timed automaton as an associated infinite-state, infinite-branching TPS.

Definition 6.6 (Semantics of probabilistic timed automata)**.** Let $\mathscr{A} = (\Sigma, Q, q_0, X, I, \rightarrow)$ be a probabilistic timed automaton. The *semantics* of $\mathscr{A}$ is the TPS $\mathscr{T}_{\mathscr{A}} = (S, S_0, \Sigma, \Rightarrow)$ with $S = \{(q, w) \in Q \times (X \rightarrow \mathbb{R}_{\geq 0}) \mid w \models I(q)\}$, $S_0 = \{(q_0, \mathbf{0})\}$ where $\mathbf{0}(x) = 0$ for all $x \in X$, and where $((q, w), d, a, \mu) \in \Rightarrow$ if both of the following conditions hold :

Time elapse: $w + d \models I(q)$;

Edge traversal: there exists a probabilistic edge $(q, g, a, \eta) \in \rightarrow$ such that $w + d \models g$ and, for each $(\rho, q') \in \mathsf{support}(\eta)$, we have $\mu(q', \rho(w + d)) = \eta(\rho, q')$.

■

The semantics of a probabilistic timed automaton is then given in terms of paths of the form: $\omega = (q_0, w_0) \xrightarrow{d_0, a_0, \mu_0} (q_1, w_1) \xrightarrow{d_1, a_1, \mu_1} \cdots \xrightarrow{d_{n-1}, a_{n-1}, \mu_{n-1}} (q_n, w_n)$. We will represent paths graphically in a similar way as for runs in the non-probabilistic framework (see Definition 2.12). More precisely, a path is represented by a directed graph where states are depicted within nodes containing the name of the location and the value of each of the clocks, and probabilistic edges are depicted using edges labeled with a triple containing the time duration, the name of the action, and the probability.

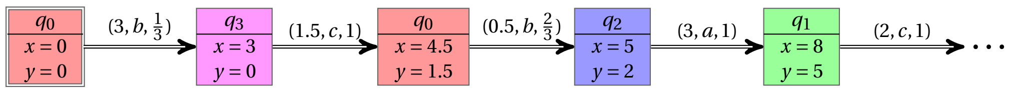


Figure 6.3: Example of path

Example 6.7. Consider again the probabilistic timed automaton $\mathscr{A}$ of Example 6.3. Then Figure 6.3 depicts an example of path for $\mathscr{A}$.

□

We write $Path^{\mathscr{A}}_{fin}$ for $Path^{\mathscr{T}_{\mathscr{A}}}_{fin}$. Observe that the rule for discrete transitions is a simplified version of the standard rule [KNSS02], which is permitted by the assumption of reset unicity. The definition of $\mathscr{T}_{\mathscr{A}}$ also relies on the fact that $\mathscr{A}$ satisfies the well-formedness assumption explained below.

In order to define the notion of well-formedness for probabilistic timed automata, we introduce below the notions of admissible target and no deadlock.

Admissible Targets. A probabilistic timed automaton has *admissible targets* if whenever a probabilistic edge is enabled, all of the probabilistic alternatives (pairs of target location and clock reset) result in valid states; that is, they do not result in pairs (q, w) in which w does not satisfy $I(q)$. We introduce more formally this notion in the following definition.

Definition 6.8 (Admissible targets)**.** A probabilistic timed automaton is said to have admissible targets if, for each probabilistic edge $(q, g, a, \eta) \in \rightarrow$ and state $(q, w) \in S$ such that $w \models g$, we require that $(q', \rho(w)) \in S$ for each $(\rho, q') \in \mathsf{support}(\eta)$. ■

Example 6.9. An example of a probabilistic timed automaton which does not have admissible targets is illustrated in Figure 6.2 (left) page 145. It is possible that the value of the clock x exceeds 3 when the lower probabilistic edge from q_0 is taken, in which case, on taking the probabilistic alternative labeled by $y := 0$, the invariant of q_2 is not satisfied. Instead, both the probabilistic timed automaton fragment on the right-hand side of Figure 6.2 and the probabilistic timed automaton of Example 6.3 have admissible targets. □

A probabilistic timed automaton can be transformed into a probabilistic timed automaton with admissible targets by incorporating the invariant associated with the target location into the guard of each probabilistic edge (along the lines of the transformation in [KNSW07]).

No Deadlock. To guarantee the existence of at least one transition from each state, we assume that $\mathscr{A}$ has *no deadlock*, as explained in the following definition [Spr01].

Definition 6.10 (No deadlock)**.** A probabilistic timed automaton $\mathscr{A}$ is said to have *no deadlock* if, in all states of $\mathscr{A}$ reachable from $(q_0, \mathbf{0})$ (i.e., final states of paths in $Path_{fin}^{\mathscr{T}_{\mathscr{A}}}$), it is always possible to take some probabilistic edge, possibly after letting time elapse. ■

This assumption guarantees that the probabilistic transition relation of the associated probabilistic system is total (see Section 6.2.1).

Example 6.11. We give in Figure 6.4 an example of probabilistic timed automaton which is not well-formed because, although it has admissible targets, it does not verify the no deadlock assumption. Indeed, after several transitions between q_0 and q_3, it may happen that the systems gets deadlocked in location q_3 with $x > 10$, because x is never reset. In that case, it is not possible to get leave q_3 to q_0 because the invariant of q_0 is not verified anymore.

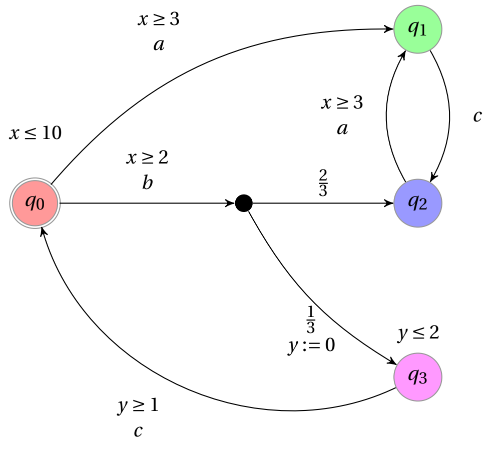


Figure 6.4: A probabilistic timed automaton containing a deadlock

□

We can now define the notion of well-formedness for probabilistic timed automata.

Definition 6.12 (Well-formedness)**.** A probabilistic timed automaton is said to be *well-formed* if it satisfies the following two assumptions:

- admissible targets, and
- no deadlock.

■

For the remainder of this chapter, we assume that all of the probabilistic timed automata we consider are well-formed.

6.2.3 Parametric Probabilistic Timed Automata

We now introduce in the following definition an extension of probabilistic timed automata to the parametric case [AFS09]. Parametric probabilistic timed automata allow the use of parameters in place of constants within guards and invariants, and are based on the parameterization of timed automata into parametric timed automata [AHV93].

Definition 6.13 (PPTA)**.** Given a set X of clocks and a set P of parameters, a *parametric probabilistic timed automaton (PPTA)* $\mathscr{A}$ is a tuple of the form $\mathscr{A} = (\Sigma, Q, q_0, X, P, I, \rightarrow)$, where:

- Σ is a finite set of *actions*,
- Q is a finite set of *locations* with an *initial location* $q_0 \in Q$,
- X is a finite set of clocks,
- P is a finite set of parameters,
- I is the *invariant* function, assigning to every $q \in Q$ a constraint $I(q)$ on the clocks X and the parameters P, and
- $\rightarrow$ is the *probabilistic edge relation* consisting of elements of the form (q, g, a, η), where $q \in Q$, g is a constraint on the clocks X and the parameters P, $a \in \Sigma$, and $\eta \in \mathsf{Dist}(2^X \times Q)$.

■

Example 6.14. We give in Figure 6.5 an example of PPTA, containing 4 locations (viz., q_0, q_1, q_2 and q_3), 2 clocks (viz., x and y) and 3 parameters (viz., p_1, p_2 and p_3)

□

Instantiation of a PPTA. Similarly to the instantiation of a PTA into a TA (see Section 2.2.3), we now define the instantiation of a PPTA into a probabilistic timed automaton. Given a PPTA $\mathscr{A} = (\Sigma, Q, q_0, X, P, I, \rightarrow)$, for every parameter valuation $\pi = (\pi_1, \dots, \pi_M)$, $\mathscr{A}[\pi]$ denotes the PPTA obtained from $\mathscr{A}$ by substituting every occurrence of a parameter p_i by constant π_i in the guards and invariants of $\mathscr{A}$. We say that p_i is *instantiated* with π_i. Formally, $\mathscr{A}[\pi] = (\Sigma, Q, q_0, X, P, I', \rightarrow')$, where I' and $\rightarrow'$ are defined in the following way: for each location $q \in Q$, we let $I'(q) = I(q)[\pi]$, and we let $\rightarrow'$ be the smallest set such that, for each $(q, g, a, \eta) \in \rightarrow$, we have $(q, g[\pi], a, \eta) \in \rightarrow'$. Note that, as all parameters are instantiated, $\mathscr{A}[\pi]$ is a standard probabilistic timed automaton.

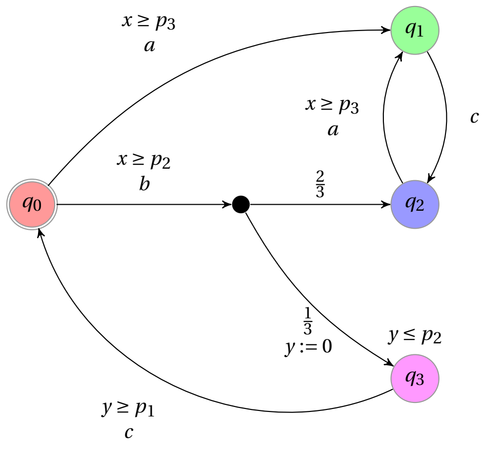


Figure 6.5: Example of PPTA

Strictly speaking, $\mathscr{A}[\pi]$ is a probabilistic timed automaton only when π assigns a natural number (rather than a real) to each parameter, but this does not matter in our context. Note also that, for each location $q \in Q$ of $\mathscr{A}[\pi]$, we have that $I'(q)$ is a constraint only on clocks, and, for each edge $(q, g, a, \eta) \in \rightarrow$, we have that g is a constraint only on clocks.

Example 6.15. Consider again the PPTA $\mathscr{A}$ of Example 6.14 and the following reference valuation π_0 of the parameters: $\pi_0 : p_1 = 1 \wedge p_2 = 2 \wedge p_3 = 3$. Then, $\mathscr{A}[\pi_0]$ corresponds to the (non-parametric) probabilistic timed automaton of Example 6.3. □

Assumptions. As for the probabilistic timed automata, we make for the PPTAs the following assumptions, as defined in Section 6.2.2 page 144.

Determinism on actions: Given a location $q \in Q$ and action $a \in \Sigma$, there is at most one probabilistic edge of the form $(q, \_, a, \_) \in \rightarrow$.

Reset unicity: For any probabilistic edge $(q, g, a, \eta) \in \rightarrow$ and location $q' \in Q$, there exists at most one $\rho \in 2^X$ such that $\eta(\rho, q') > 0$.

Well-formedness of PPTAs. We say that a PPTA $\mathscr{A}$ is well-formed if, for each parameter valuation π, the resulting probabilistic timed automaton $\mathscr{A}[\pi]$ is well-formed, as defined in Definition 6.12. For the remainder of this chapter, we assume that all of the PPTAs we consider are well-formed.

Network of PPTAs. Networks of PPTAs can be defined by using parallel composition based on the synchronization of discrete transitions of different components sharing the same action in a very similar manner to networks of probabilistic timed automata (see Definition 6.5).

Trace distributions

Let $\mathscr{A} = (\Sigma, Q, q_0, X, P, I, \rightarrow)$ be a PPTA, and let π be a valuation of the parameters in P. Given a path $\omega = (q_0, w_0) \xrightarrow{d_0, a_0, \mu_0} (q_1, w_1) \xrightarrow{d_1, a_1, \mu_1} \cdots \xrightarrow{d_{n-1}, a_{n-1}, \mu_{n-1}} (q_n, w_n)$ of $\mathscr{A}[\pi]$, we let the *time-abstract trace* of ω be the sequence of alternating locations and actions $q_0 a_0 q_1 a_1 \cdots a_{n-1} q_n$. In the remainder of this chapter, for the sake of simplicity and coherence with the traces defined in the non-probabilistic framework (see Definition 2.14), we will simply refer to time-abstract traces as *traces*.

As for the non-probabilistic framework, we depict traces under a graphical form using boxed nodes labeled with locations and double arrows labeled with actions.

Example 6.16. Consider again the path depicted in Example 6.7. Then the associated trace is the one depicted in Figure 6.6.

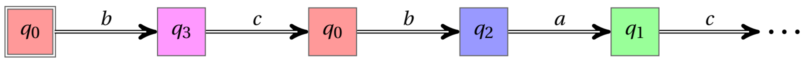


Figure 6.6: Example of trace

□

Given a scheduler σ, we let $\mathsf{trace}^\sigma : \mathit{Path}^\sigma_{\mathit{fin}} \rightarrow (Q \times \Sigma)^*$ be the function associating the trace with each path of $\mathit{Path}^\sigma_{\mathit{fin}}$. Then the *(time-abstract) trace distribution* of σ and state $s \in S$ is the probability measure over traces denoted by td^σ_s defined according to trace^σ and the trace distribution construction of Segala [Seg95]. Although we do not consider the details of the construction of trace distributions in this thesis, we note that, for example, the probability assigned by td^σ_s to traces in which a certain location is reached is defined to be the same as the probability assigned by Prob^σ_s to the set of paths in which this location is reached. The set of trace distributions of the TPS $\mathscr{T}_{\mathscr{A}[\pi]}$ is denoted by $\mathsf{tdist}(\mathscr{T}_{\mathscr{A}[\pi]}) = \{\mathsf{td}^\sigma_s \mid \sigma \text{ is a scheduler of } \mathscr{T}_{\mathscr{A}[\pi]} \text{ and } s \in S_0\}$.

Let π and π' be two valuations of the parameters in P. We say that $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi']$ are *(time-abstract) trace distribution equivalent*, written $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$, if $\mathsf{tdist}(\mathscr{T}_{\mathscr{A}[\pi]}) = \mathsf{tdist}(\mathscr{T}_{\mathscr{A}[\pi']})$. If $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$, we can conclude that the TPSs have time-abstract equivalent finite behaviors: for example, they assign the same maximum and minimum probabilities of reaching a certain location [KNS02]. Actually, more generally, they assign the same maximum and minimum probabilities to linear-time properties on finite traces.

We now recall from [KNS02, KNS03] a sufficient condition for guaranteeing trace distribution equivalence between $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi']$, which will be useful in Section 6.4. First we introduce the notion of time-abstract path equivalence.

Definition 6.17. The path $\omega = (q_0, w_0) \xrightarrow{d_0,a_0,\mu_0} \cdots \xrightarrow{d_{n-1},a_{n-1},\mu_{n-1}} (q_n, w_n)$ of $\mathscr{T}_{\mathscr{A}[\pi]}$, is said to be *time-abstract path equivalent* to the path $\omega' = (q'_0, w'_0) \xrightarrow{d'_0,a'_0,\mu'_0} \cdots \xrightarrow{d'_{n-1},a'_{n-1},\mu'_{n-1}} (q'_n, w'_n)$ of $\mathscr{T}_{\mathscr{A}[\pi']}$, written $\omega \equiv \omega'$, if $q_i = q'_i$, $a_i = a'_i$, and $\mu_i(q_{i+1}, w_i) = \mu'_i(q'_{i+1}, w'_i)$ for all $i = 0,\ldots,n-1$, and $q_n = q'_n$.

We extend the notion of time-abstract path equivalence to sets of paths: two sets of paths $\Omega \subseteq Path_{fin}^{\mathscr{A}[\pi]}$ and $\Omega' \subseteq Path_{fin}^{\mathscr{A}[\pi']}$ are time-abstract path equivalent, written $\Omega \equiv \Omega'$, if

1. for each path $\omega \in \Omega$, there exists $\omega' \in \Omega'$ such that $\omega \equiv \omega'$, and

2. conversely, for each path $\omega \in \Omega'$, there exists $\omega' \in \Omega$ such that $\omega \equiv \omega'$.

■

The following result from [KNS02, KNS03] allows us to relate time-abstract equivalence on paths to (time-abstract) trace distribution equivalence. This proposition states that time-abstract path equivalence implies also trace distribution equivalence.

Proposition 6.18. *Let $\mathscr{A}$ be a PPTA, and let π and π' be valuations of parameters P. If $Path_{fin}^{\mathscr{A}[\pi]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}[\pi']}(q_0, \mathbf{0})$, then $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$.*

Non-probabilistic Version of a PPTA

In this subsection, along the lines of [KNS02, KNS03], we explain how probability values can be abstracted away from a PPTA to result in a classical, non-probabilistic parametric timed automaton. First we explain how a PPTA can be transformed into a PPTA featuring point distributions only. This is done by replacing probabilistic choice within a single probabilistic edge by nondeterministic choice between multiple probabilistic edges, each of which corresponds to a point distribution.

Definition 6.19 (Non-probabilistic version of a PPTA)**.** Let $\mathscr{A}$ be a PPTA of the form $(\Sigma, Q, q_0, X, P, I, \rightarrow)$. The *non-probabilistic version* of $\mathscr{A}$, written $\mathscr{A}^* = (\Sigma, Q, q_0, X, P, I, \rightarrow^*)$, is a PPTA which agrees with $\mathscr{A}$ on all elements apart from the probabilistic edge relation: let $\rightarrow^*$ be the smallest probabilistic edge relation such that for every edge $(q, g, a, \eta) \in \rightarrow$ and $(\rho, q') \in \text{support}(\eta)$, we have $(q, g, a, \eta_{(\rho,q')}) \in \rightarrow^*$. ■

Recall that, in the previous definition, $\eta_{(\rho,q')}$ denotes the point distribution assigning probability 1 to the element (ρ, q').

We note that a PPTA featuring point distributions only has a one-to-one mapping with a classical parametric timed automaton: a probabilistic edge $(q, g, a, \eta_{(\rho,q')})$ of the PPTA corresponds to the edge (q, g, a, ρ, q') of a classical parametric timed automaton. In subsequent sections of this chapter, this will allow us to apply methods for classical parametric timed automata to PPTAs featuring point distributions.

Example 6.20. Consider again the PPTA $\mathscr{A}$ of Example 6.14. Then the non-probabilistic version $\mathscr{A}^*$ of $\mathscr{A}$ is the non-probabilistic parametric timed automaton depicted in Figure 6.7.

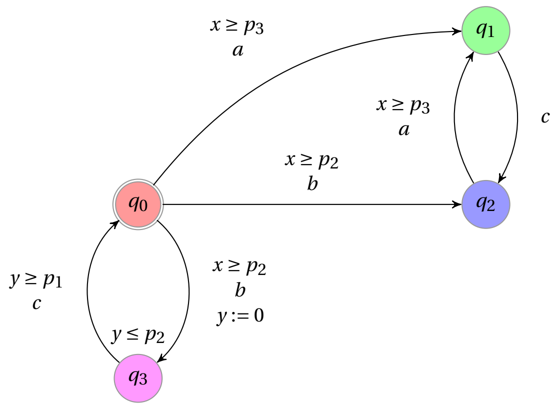


Figure 6.7: Example of non-probabilistic version of a PPTA

□

Observe that the state sets of $\mathscr{T}_{\mathscr{A}[\pi]}$ and $\mathscr{T}_{\mathscr{A}^*[\pi]}$ are equal.

Proposition 6.21. *Let π be a valuation of P and (q, w) be a state of $\mathscr{T}_{\mathscr{A}[\pi]}$ (and $\mathscr{T}_{\mathscr{A}^*[\pi]}$). For each step $(q, w) \xrightarrow{d,a,\mu} (q', w')$ of $\mathscr{T}_{\mathscr{A}[\pi]}$, there exists the step $(q, w) \xrightarrow{d,a,\mu_{(q',w')}} (q', w')$ of $\mathscr{T}_{\mathscr{A}^*[\pi]}$. Conversely, for each step $(q, w) \xrightarrow{d,a,\mu_{(q',w')}} (q', w')$ of $\mathscr{T}_{\mathscr{A}^*[\pi]}$, there exists a step $(q, w) \xrightarrow{d,a,\mu} (q', w')$ of $\mathscr{T}_{\mathscr{A}[\pi]}$.*

Proposition 6.21 allows us to obtain a one-to-one mapping between transitions of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$. By reasoning inductively, we can extend the proposition to obtain a one-to-one mapping between paths of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$. Note that, by the combination of determinism on actions and reset unicity, the probability of the transitions of $\mathscr{A}[\pi]$ is encoded in the actions and target locations of the associated transitions of $\mathscr{A}^*[\pi]$. This, together with the one-to-one mapping between paths of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$, implies that, for any pair ω_*, ω'_* of paths such that $\omega_* \in Path_{fin}^{\mathscr{A}^*[\pi]}(q_0, \mathbf{0})$, $\omega'_* \in Path_{fin}^{\mathscr{A}^*[\pi']}(q_0, \mathbf{0})$ and $\omega_* \equiv \omega'_*$, we can generate the paths ω, ω', such that $\omega \in Path_{fin}^{\mathscr{A}[\pi]}(q_0, \mathbf{0})$, $\omega' \in Path_{fin}^{\mathscr{A}[\pi']}(q_0, \mathbf{0})$ and $\omega \equiv \omega'$. Together, these facts allow us to show that, given a PPTA $\mathscr{A}$ and π and π' valuations of the parameters, if the paths of the non-probabilistic version of $\mathscr{A}$ are equivalent for π and π', then the paths are also equivalent in $\mathscr{A}$ for π and π'. This is formalized in the following proposition.

Proposition 6.22. *Let $\mathscr{A}$ be a PPTA, and let π and π' be valuations of the parameters. If $Path_{fin}^{\mathscr{A}^*[\pi]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}^*[\pi']}(q_0, \mathbf{0})$, then $Path_{fin}^{\mathscr{A}[\pi]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}[\pi']}(q_0, \mathbf{0})$.*

It is on this proposition that the extension of the inverse method to the probabilistic framework relies.

6.3 The Inverse Problem for PPTAs

We can now state formally the inverse problem for PPTAs, which was sketched in Section 6.1.

The Problem. Given a PPTA $\mathscr{A}$ and a valuation π_0 of the parameters, we present in the following a method allowing to synthesize a constraint K_0 on the parameters of $\mathscr{A}$ such that $\pi_0 \models K_0$ and, for all $\pi \models K_0$, $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$ are trace distribution equivalent. As a consequence, they assign the same maximum and minimum probabilities to linear-time properties on finite traces.

More formally, the problem can be stated as follows [AFS09].

The Inverse Problem for PPTAs
Let $\mathscr{A}$ a PPTA and π_0 a valuation of the parameters. Find a constraint K_0 such that :

1. $\pi_0 \models K_0$, and
2. $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$, for all $\pi \models K_0$.

In Chapter 3, we introduced the *inverse method algorithm IM*, which allows us to solve the inverse problem in the non-probabilistic framework. Following the notations of this chapter, the result of the algorithm *IM* can be rewritten as follows: Given a (non-probabilistic) parametric timed automaton $\mathscr{A}$ and a reference valuation π_0, $IM(\mathscr{A}, \pi_0)$ synthesizes a constraint K_0 such that

1. $\pi_0 \models K_0$, and
2. $Path_{fin}^{\mathscr{A}[\pi_0]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}[\pi]}(q_0, \mathbf{0})$ for all $\pi \models K_0$.

We now show how the results of Chapter 3 can be extended to the synthesis of parameters preserving the value of minimum and maximum probabilities for reachability properties.

6.4 Extension of the Inverse Method

Principle. Given a PPTA $\mathscr{A}$, we can now solve the inverse problem for $\mathscr{A}$ by applying the algorithm *IM* to the non-probabilistic version $\mathscr{A}^*$ of $\mathscr{A}$ [AFS09].

Properties. We state below the correctness of our method.

Theorem 6.23 (Correctness)**.** *Given a PPTA $\mathscr{A}$ and a reference valuation π_0, the constraint K_0 returned by IM$(\mathscr{A}^*, \pi_0)$ solves the inverse problem for PPTAs, i.e.:*

1. $\pi_0 \models K_0$*, and*
2. $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$*, for all* $\pi \models K_0$.

Proof. Since K_0 is a solution of the inverse problem for $\mathscr{A}^*$, we have $Path_{fin}^{\mathscr{A}^*[\pi]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}^*[\pi_0]}(q_0, \mathbf{0})$ for all $\pi \models K_0$; hence, we have by Proposition 6.22 that $Path_{fin}^{\mathscr{A}[\pi]}(q_0, \mathbf{0}) \equiv Path_{fin}^{\mathscr{A}[\pi_0]}(q_0, \mathbf{0})$ for all $\pi \models K_0$. From Proposition 6.18, we conclude that $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$ for all $\pi \models K_0$. ■

Application to the Computation of Probabilities. Given a PPTA $\mathscr{A}$ and a valuation π_0 of the parameters, in order to determine the minimum (resp. maximum) probability *pr* of reaching a given location of $\mathscr{A}[\pi_0]$, it is sufficient to proceed as follows:

1. Compute $K_0 = IM(\mathscr{A}^*, \pi_0)$;
2. Compute *pr* (using, e.g., PRISM) for $\mathscr{A}[\pi_1]$, for some $\pi_1 \models K_0$.

As a consequence of Theorem 6.23, given the computation of K_0 using $IM(\mathscr{A}^*, \pi_0)$, the minimum and maximum probabilities of satisfying linear-time properties on finite traces will be the same in $\mathscr{A}[\pi_1]$ and $\mathscr{A}[\pi_0]$.

An advantage of our method is that one can take π_1 small enough in order to make the computation of PRISM easier, because the performance of PRISM depends on the size of the state space of the model used as input, which in turn depends on the size of the constants used in the PTA (see Section 6.5 for a comparison for various case studies).

Remark 6.24. The constraint K_0 output by our method is *not* (in general) the weakest constraint satisfying the inverse problem as defined in Section 6.3. One reason for this is that the constraint output by the inverse method defined in Chapter 3 for (non-probabilistic) parametric timed automata is always in conjunctive form. In contrast, the weakest constraint may be in disjunctive form (see Proposition 3.13 and the following remarks on the non-maximality of the inverse method in the non-probabilistic framework). We will address this problem in Section 6.6. □

6.5 Case Studies

In this section, we show the interest of the inverse method in the context of three case studies. More precisely, we consider three protocols, each modeled as a PPTA, and each with a reference valuation π_0 taken from the associated standard.[1]

Our approach consists of the following two phases:

1. Using the tool IMITATOR II which implements the inverse method in the non-probabilistic framework (see Section 4.1.2), we synthesize a constraint K_0 for the *non-probabilistic* version of the protocol.

2. Using the probabilistic model-checking tool PRISM [HKNP06, pWpb], we compute minimum/maximum reachability probabilities for various properties with regard to a number of parameter valuations satisfying or not K_0. For parameter valuations satisfying K_0, the probabilities computed by PRISM are equal (as stated by Theorem 6.23); we also compute the probabilities for some parameter valuations not satisfying K_0.

Experiments were performed on an Intel Core 2 Duo with 2GB of RAM. We will consider the following three randomized protocols:

1. the IEEE 1394 Root Contention Protocol (Section 6.5.1);

[1]Note that we consider acyclic versions of those protocols (roughly speaking, by bounding the number of rounds in Section 6.5.1, and by bounding the maximal number of collisions in Section 6.5.2 and Section 6.5.3).

2. the CSMA/CD Protocol (Section 6.5.2);

3. the IEEE 802.11 Wireless Local Area Network Protocol (Section 6.5.3).

6.5.1 Root Contention Protocol

Consider again the Root Contention Protocol introduced in the probabilistic framework in Section 6.1. We consider the probabilistic timed automata model of [KNS03]. We give in Figure 6.8 the PPTA model of node i.

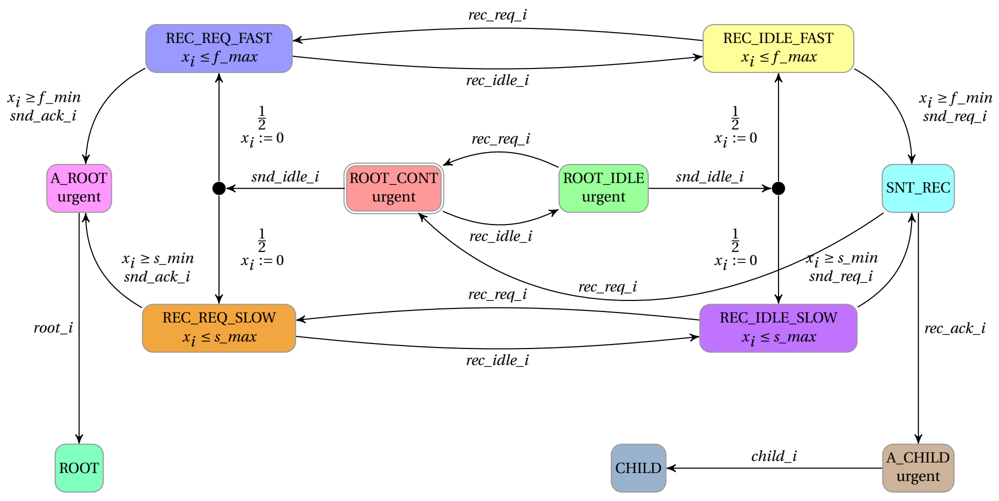

rec_req_i
REC_REQ_FAST $x_i \leq f\_max$
REC_IDLE_FAST $x_i \leq f\_max$
rec_idle_i
$x_i \geq f\_min$ snd_ack_i
$\frac{1}{2}$ $x_i := 0$
$x_i \geq f\_min$ snd_req_i
A_ROOT urgent
snd_idle_i
ROOT_CONT urgent
rec_req_i
ROOT_IDLE urgent
snd_idle_i
SNT_REC
rec_idle_i
$x_i \geq s\_min$ snd_ack_i
$x_i \geq s\_min$ snd_req_i
rec_req_i
root_i
REC_REQ_SLOW $x_i \leq s\_max$
REC_IDLE_SLOW $x_i \leq s\_max$
rec_ack_i
ROOT
CHILD
child_i
A_CHILD urgent

Figure 6.8: PPTA modeling node i in the Root Contention Protocol

Due to the absence of probabilities in the wire, the model of the wire actually corresponds to the one given in Figure 4.9 page 93 in the non-probabilistic framework.

Also recall the reference valuation π_0:

$$\begin{array}{lll} f\_min = 760 & f\_max = 850 & delay = 360 \\ s\_min = 1590 & s\_max = 1670 & \end{array}$$

Let us now consider a *non-probabilistic* model of this protocol. The non-probabilistic version of the PPTA modeling the node (given in Figure 6.8) is the PTA given in Figure 4.8 page 92. The non-probabilistic model of the wire remains the one given in Figure 4.9 page 93.

Recall from Section 4.4 that, by applying IMITATOR II to this non-probabilistic parametric timed automaton version of the protocol and the reference valuation π_0, we synthesize the following constraint in about 2 s:

$$K_0 : 2delay < rc\_fast\_min \ \wedge \ rc\_fast\_max + 2delay < rc\_slow\_min.$$

Recall also that this constraint is the same as the one synthesized in [HRSV02]. We now consider the following properties:

- $Prop_3$: minimum probability that a leader is elected after 3 rounds or less.
- $Prop_5$: minimum probability that a leader is elected after 5 rounds or less.

Name	*f_max*	*f_min*	*s_max*	*s_min*	*delay*	$\models K_0$	States	Constr.	$Prop_3$	$Prop_5$	$= \pi_1$
π_0	850	760	1670	1590	360	yes	-	-	-	-	-
π_1	85	76	167	159	36	yes	857631	41	47	124	yes
π_2	4	3	8	7	1	yes	1168	0.2	0.1	0.2	yes
π_3	85	76	167	159	37	no	915926	40	49	121	no

Table 6.1: Results of PRISM for the RCP

The results of the application of PRISM to different parameter valuations are given in Table 6.1. The second to sixth columns give the considered parameter values. The seventh column indicates whether the parameter valuation satisfies K_0, while the eighth and ninth columns give the number of states corresponding to the valuation, together with the time in seconds required by PRISM for the construction of the model. The following two columns give the time in seconds used by PRISM to perform verification of the two considered properties.[2] For the parameter valuations π_2 and π_3, the final column indicates whether the probabilities computed for the properties are the same as those computed for π_1.

For the reference valuation π_0, PRISM did not terminate after 2 hours. The probabilities for $Prop_3$ and $Prop_5$ are 0.75 and 0.94, respectively, for all parameter valuations except π_3, which does not satisfy K_0. In the case of π_3, the probabilities of the two properties are 0.63 and 0.79, respectively. The parameter valuation π_1 is an exact rescaling of π_0; thus it is trivial to see that the parameter valuations π_0 and π_1 result in equivalent models. However, we note that π_2 obtained from K_0 is associated with a significantly smaller state space, which leads to significantly lower analysis times, than π_1.

6.5.2 CSMA/CD Protocol

We now apply our method to the CSMA/CD Protocol, as studied in the context of probabilistic timed automata in [KNSW07]. We consider the case when there are two stations, 1 and 2, trying to send data at the same time. The overall model is given by the parallel composition of three probabilistic timed automata representing the medium and two stations trying to send data.

[2]The verification engine used was the sparse matrix engine.

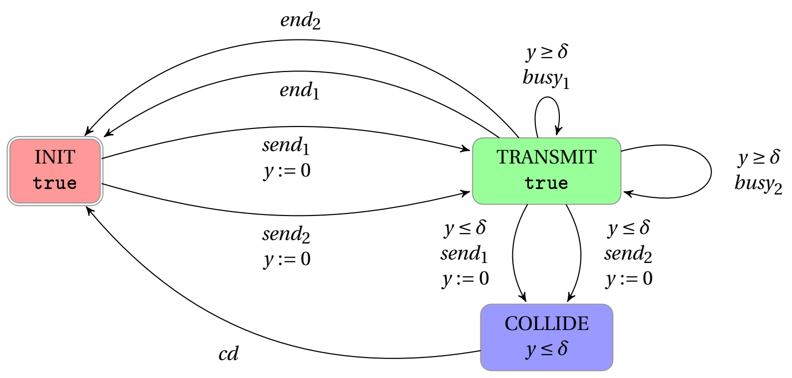


Figure 6.9: CSMA/CD Medium

The probabilistic timed automaton representing the medium is given in Figure 6.9. The medium is initially ready to accept data from any station (event $send_1$ or $send_2$). Once a station, say 1, starts sending its data there is an interval of time (at most δ), representing the time it takes for a signal to propagate between the stations[3]. When a collision occurs, there is a delay (again at most δ) before the stations realize there has been a collision, after which the medium will become free. If the stations do not collide, then when station 1 finishes sending its data (event end_1) the medium becomes idle.

The probabilistic timed automaton representing a station i ($i = 1,2$) is given in Figure 6.10. Station i starts by sending its data. If there is no collision, then, after λ time units, the station finishes sending its data (event end_i). On the other hand, if there is a collision (event cd), the station attempts to retransmit the packet. The delay before retransmitting is a *random* integer number of time slots (each of length *slot*). More precisely, the number of slots that station i waits after the nth transmission failure is chosen as a uniformly distributed *random* integer in the range: $0,1,2,\dots,2^{bc_i+1}-1$, where $bc_i = \min(n, bcmax)$, and $bcmax$ is a fixed upper bound for bc_i (initially: $bc_i = n = 0$). This random choice is depicted in Figure 6.10 by the assignment $backoff_i := \mathbf{RAND}(bc_i) * slot$. Once this time has elapsed, if the medium appears free the station resends the data (event $send_i$), while if the medium is sensed busy (event $busy_i$) the station repeats this process.

We consider the following reference valuation π_0 taken from the IEEE standard 802.3 for 10 Mbps Ethernet:

$$\lambda = 808\mu s \quad slot = 52\mu s \quad \delta = 26\mu s.$$

[3]In the literature, the propagation time between the stations is usually denoted by variable σ. We choose here to make use of δ in order not to create any confusion with the schedulers that we name σ in this chapter.

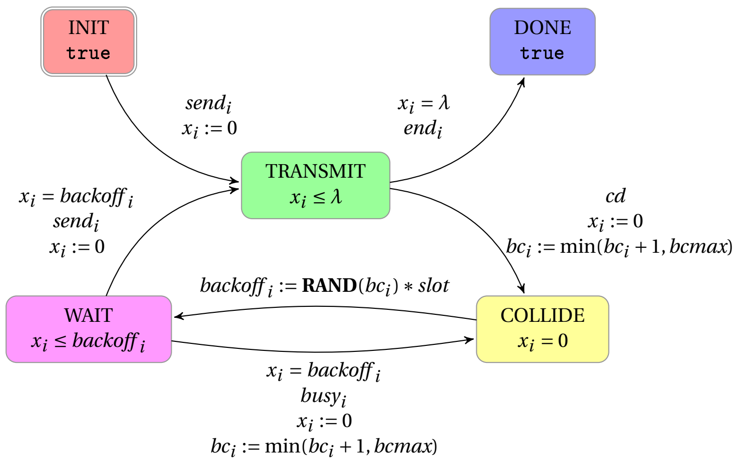


Figure 6.10: CSMA/CD Station i

As described in Section 6.4, from a PPTA describing this system, we can then obtain a non-probabilistic parametric timed automaton. Applying IMITATOR II to this non-probabilistic model and the reference valuation π_0, we obtain the following constraint:

$$K_0 : \ 0 < \delta < slot \ \wedge \ 15slot < \lambda < 16slot.$$

This constraint is such that $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$ are trace-distribution equivalent, for any $\pi \models K_0$. We consider the following three properties:

- $Prop_j$, for $j \in \{1,2\}$: minimum probability that station 1 transmits its message after exactly j collisions.
- $Prop_{\leq 3}$: minimum probability that station 1 transmits its message with no more than 3 collisions.

We apply PRISM to the system with the parameters set to different valuations (including π_0). The probabilities corresponding to the three properties, for the reference valuation π_0, are 0.5, 0.38 and 0.97, respectively. From the results of Section 6.4, the probabilities for π_0, π_1 and π_2 are identical. Instead, for π_3, in which the constraint K_0 is violated by considering the limit case where $\delta = slot$, the probabilities of the properties are 0, 0.19 and 0.61, respectively. A further observation is that, even if the value of λ violates K_0 (this is the case of π_4), the probabilities can remain the same as for π_0. Indeed, as noted in Section 3.3.3, the constraint synthesized by the algorithm *IM* is not necessarily the weakest.

Name	λ	$slot$	δ	$\models K_0$	States	Constr.	$Prop_1$	$Prop_2$	$Prop_{\leq 3}$	$= \pi_0$
π_0	808	52	26	yes	36335	17	6	10	14	-
π_1	31	2	1	yes	1746	0.1	0.1	0.1	0.1	yes
π_2	940	60	59	yes	42260	32	11	15	21	yes
π_3	940	60	60	no	42753	37	10	16	25	no
π_4	52	52	26	no	6212	2	0.9	2	3	yes

Table 6.2: Results of PRISM for the CSMA/CD

Additional information is given in Table 6.2. The significance of the columns is similar to that for Table 6.1. For the parameter valuations π_1 to π_4, the final column indicates whether the probabilities computed for the properties are the same as those computed for π_0.

6.5.3 Wireless Local Area Network Protocol

We also applied our method to the IEEE 802.11 Wireless Local Area Network Protocol, considering the following valuation π_0 of the parameters corresponding to the IEEE 802.11 standard and given, e.g., in [pWpb, KNS02]. (Timing values are given in μs.)

$$\begin{array}{rrr} ASLOTTIME = 50 & DIFS = 128 & VULN = 48 \\ TTMAX = 15717 & TTMIN = 224 & ACKTO = 300 \\ ACK = 205 & SIFS = 28 & \end{array}$$

In order to apply the inverse method, we have to use for this particular example the first version of IMITATOR (see Section 4.1.1), and not IMITATOR II. Indeed, the new version IMITATOR II does not allow the use of *urgent actions* (or "as soon as possible" actions), i.e., actions that the system *must* take as soon as the guards of the different transitions synchronizing on this action are all verified. On the contrary, IMITATOR is based on HYTECH, which allows the use of such actions. The implementation of this feature to IMITATOR II is the subject of future work (see Section 8.2).

Taking a parametric timed automaton version of the model and the parameter valuation π_0 as input, the tool IMITATOR computes the following constraint K_0 in about 7 hours:

$$\begin{array}{lll} & SIFS + ACK < 6ASLOTTIME & \wedge \quad 0 < VULN \\ \wedge & DIFS < 3ASLOTTIME & \wedge \quad 0 < SIFS \\ \wedge & 2ASLOTTIME < DIFS & \wedge \quad 0 < ACK \\ \wedge & VULN < ASLOTTIME & \wedge \quad SIFS < DIFS \\ \wedge & 5ASLOTTIME < VULN + DIFS + TTMIN & \wedge \quad TTMIN \leq TTMAX \\ \wedge & 6ASLOTTIME = ACKTO & \end{array}$$

We consider the maximum probability that either station's collision counter reaches k, for $k = 2, 3$, as considered in [KNS02].

Name	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	States	Constr.	$Prop_2$	$Prop_3$
π_0	50	128	48	15717	224	300	205	28	-	-	-	-
π_1	50	128	48	224	224	300	205	28	1671933	250	750	898
π_2	50	128	48	301	224	300	205	28	1751320	273	799	953
π_3	50	128	48	75	75	300	205	28	1527254	204	721	786
π_4	2	5	1	5	5	12	1	1	117510	0.97	3.59	4.19
π_5	4	10	2	10	10	24	2	2	169881	1.97	7.44	8.25
π_6	2	5	1	629	5	12	1	1	760854	27	27	25

Table 6.3: Results of PRISM for the IEE 802.11 Protocol

The results of the application of PRISM are given in Table 6.3, where the properties are denoted by $Prop_k$ for $k = 2, 3$. The parameters $p_1, p_2, \ldots, p_8$ stand for *ASLOTTIME*, *DIFS*, *VULN*, *TTMAX*, *TTMIN*, *ACKTO*, *ACK*, *SIFS* respectively. The significance of the columns is similar to that for Table 6.1. Note that all the parameter valuations considered satisfy K_0, and therefore the computed probabilities for all parameter valuations are the same: The probabilities for $Prop_2$ and $Prop_3$ are 0.0625 and 0.001953125, respectively.

The constraint K_0 establishes that the value of the probabilities associated with $Prop_k$, $k = 1, 2$, is insensitive to important variations of *TTMAX*, i.e., parameter p_4 (provided its value remains greater or equal to *TTMIN*, i.e, $p_4 \geq p_5$). Although PRISM is unable to build the model for the original valuation π_0, we are able to compute the probabilities when rescaling down p_4 (compare, e.g., the results for π_0 and π_1). Also note that π_4 corresponds to one of the smallest possible integer valuations according to K_0: the computation time in this case is dramatically decreased compared to, e.g., π_1, thus showing the interest of our method. Finally note that π_6 gives an idea of what the effect of a "realistic" approximation of *TTMAX*, using 25 as the time unit (here $p_4 = 629$ corresponds to 15717/25 rounded up, and $p_1 = 2$ corresponds to 50/25), has on the size on the model, even if other parameters are kept as in the lowest valuation (viz., π_4).

Remark 6.25. We consider here a model with a maximal exponential backoff of $BOFF = 1$, whereas the model given in [pWpb, KNS02] considers that $BOFF = 6$. The inverse method and its implementations IMITATOR and IMITATOR II are in general not much sensitive to the size of the constants of the model. However, for this particular protocol, the waiting time before a retransmission is modeled by a nondeterministic choice between 2^{BOFF} transitions leading to 2^{BOFF} different loops. Each of those loops can be repeated up to $2^{BOFF} * ASLOTTIME$ times, thus leading to a dramatic explosion of the number of states. This, together with the fact that IMITATOR is by far less efficient than IMITATOR II (see comparison in Table 4.4 page 112), explains the much higher computation time (7 hours instead of a few seconds for the two other case studies) of K_0. For bigger values of *BOFF*, IMITATOR does not succeed to synthesize a constraint. □

6.6 Cartography of PPTAs

In this section, we address the following weakness of the inverse method applied to probabilistic timed automata: Given a PPTA $\mathscr{A}$ and a valuation π_0, the constraint $K_0 = IM(\mathscr{A}^*, \pi_0)$ may not be the largest set of parameter valuations solving the inverse problem.

We presented in Chapter 5 the behavioral cartography algorithm iterating the inverse method in the framework of non-probabilistic parametric timed automata. Recall that this algorithm allows us to cover (part of) the parametric space with *behavioral tiles*, i.e., constraints for which the sets of traces are uniform.

In this section, we extend this algorithm to the probabilistic case. We first describe the extension of Algorithm *BC* to the probabilistic framework (Section 6.6.1), and then give an application to the Root Contention Protocol (Section 6.6.2).

6.6.1 Principle of the Extension

Using the Behavioral Cartography Algorithm and the application of the inverse method to PPTAs described in Section 6.4, we can construct a cartography of a PPTA $\mathscr{A}$. This can be done in a straightforward manner by applying the algorithm *BC* to the non-probabilistic version $\mathscr{A}^*$ of $\mathscr{A}$.

From Theorem 6.23, we have the following proposition.

Proposition 6.26. *Let $\mathscr{A}$ be a PPTA and let V_0 be a rectangle. Let Tiling = $BC(\mathscr{A}^*, V_0)$. Then for each tile $K \in$ Tiling, for all $\pi, \pi' \models K$, $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$.*

Given a reachability property, one can then construct a *probabilistic cartography* of the system. Formally, given a PPTA $\mathscr{A}$, a rectangle V_0 and a reachability property *prop*:

1. Compute *Tiling* $= BC(\mathscr{A}^*, V_0)$;
2. For each tile $K \in$ *Tiling*, select $\pi \models K$, and compute the minimum (resp. maximum) probability of satisfying *prop* in $\mathscr{A}[\pi]$ (using, e.g., PRISM).

Discussion. An advantage of the cartography algorithm is that, if one wants to consider another reachability property *prop*′, one can keep *Tiling* as computed in step 1, and only needs to redo step 2. Only the value of the considered probability in each tile changes, leading to different partitions into good and bad subspaces.

Note also that, as we proposed in general for the inverse method, for the sake of efficiency when using PRISM, one should rather select a "small" π for each tile K (i.e., a valuation with small constants).

It is perfectly possible that several tiles correspond to the same trace sets. Indeed, the constraint synthesized by each call to *IM* is not necessarily the maximal one as stated in Proposition 3.13. In that case, it is possible to group several tiles into a single one (possibly non convex). As a consequence, it is enough to compute a probability for only one of those tiles, and one can apply it to the different tiles having the same trace sets. This is in particular interesting when one wants to compute values of probabilities for a lot of different properties. In that case, the cost induced by testing the equality of trace sets may be smaller than the gain of computing probabilities for fewer tiles.

6.6.2 Example: Root Contention Protocol

We compute a cartography of the Root Contention Protocol as described in Section 6.5.1 using the following rectangle V_0:

$$s\_min \in [140, 200] \quad s\_max \in [140, 200] \quad delay \in [1, 50].$$

The two other parameters remain constant, i.e., $f\_min = 76$ and $f\_max = 85$. Note that, in order to reduce the number of points to be covered by the algorithm, we divided by 10 the reference valuation π_0 of the parameters given in Section 6.5.1. This is equivalent to calling the inverse method only on the integer points which are multiples of 10 instead of on all integer points.

Using IMITATOR II, we compute the cartography given in Figure 6.11. Recall that we gave in Figure 5.2 page 122 this cartography as automatically output by IMITATOR II. For the sake of clarity, we project onto *delay* and *s_min*. In each tile, the parameter *s_max* is only bound by the implicit constraint $s\_min \leq s\_max$.

Remark 6.27. Tiles 1 and 6 are infinite towards dimension *s_min*, and all tiles are infinite towards dimension *s_max*. Moreover, although all the integer valuations within V_0 are covered (from the algorithm), the real-valued part of V_0 is not fully covered, because there are some "holes" (real-valued zones without integer valuations) in the lower right corner. An example of point which is not covered by the cartography is $delay = 50$, $s\_min = 140.4$ and $s\_max = 141$. □

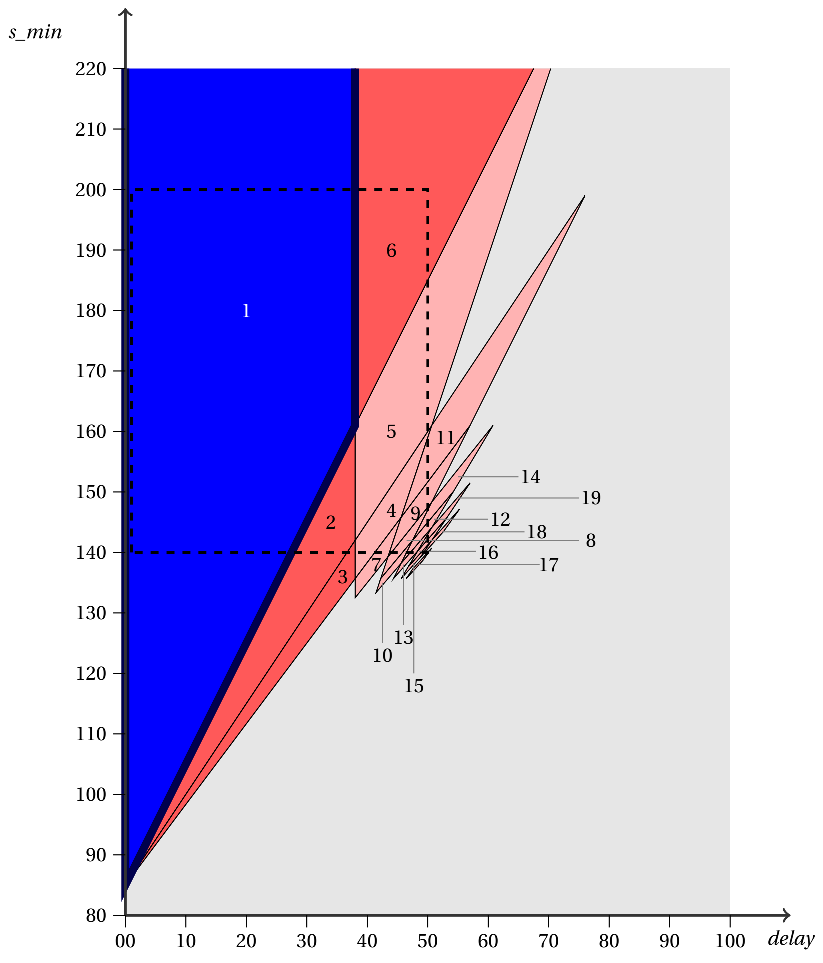


Figure 6.11: Behavioral cartography of the Root Contention Protocol according to *delay* and *s_min*

First property. We are first interested in computing the minimum probability pr_1 of satisfying the property that a leader is elected after *three* rounds or less. Using PRISM, we compute $pr_1 = 0.75$ for tile 1, $pr_1 = 0.625$ for tiles 2, 3 and 6, and $pr_1 = 0.5$ for the other tiles.

Let us suppose that a tile is good when the probability $pr_1 \geq 0.7$, and bad otherwise. In this case, the good subspace is only made of tile 1, depicted in blue in Figure 6.11. Note that, in Figure 6.11, two tiles with the same color have the same probability pr_1.

Second property. We are now interested in computing the minimum probability pr_2 of satisfying the property that a leader is elected after *five* rounds or less. We do not need to compute the cartography again, but only the value of

pr_2 for one valuation in each tile. We get $pr_2 = 0.936$ for tile 1, $pr_2 = 0.789$ for tiles 2 and 3, $pr_2 = 0.664$ for tile 6, and $pr_2 = 0.5$ for the other tiles.

Let us suppose that a tile is good when the probability $pr_2 \geq 0.7$, and bad otherwise. In this case, the good subspace is made of tiles 1, 2 and 3. (For the sake of concision, we do not give the new coloring of the cartography given in Figure 6.11.)

6.7 Related Work

Probabilistic Timed Automata. Probabilistic Timed Automata were introduced in [GJ95, KNSS02]. In this thesis, we used the notation for probabilistic timed automata presented in [KNSS02]. This model has similarities with other frameworks for probabilistic real-time systems. The approach of [GJ95] is also to augment timed automata with discrete probability distributions; however, these distributions are obtained by normalization of edge-labeling costs.

A dense time, automata-based model with discrete and continuous probability distributions is presented in [ACD91]. However, this model does not permit any nondeterministic choice, and its use of continuous probability distributions, while a highly expressive modeling mechanism, does not permit the model to be automatically verified against logics which include bounds on probability.

In [Bea03], a further model of probabilistic timed automata is introduced. The main difference with the model introduced in [KNSS02], beside the existence of acceptance locations, is the existence of a "trap" location. This allows to define Büchi acceptance conditions. The author provides in particular an algorithm of model-checking for properties of the form "is there a policy which realizes a correct behavior of the system with probability at least p".

Rescaling of Constants. Our approach provides us with a justification of the rescaling done in [KNS02], consisting in reducing the time scale of the model. However, our approach improves on the rescaling approach of [KNS02] in the following way. First recall that, in [KNS02], the rescaling is done as follows: constants appearing as an upper bound in an inequality are rounded up, and constants appearing as a lower bound are rounded down. This results in an overapproximation of the model. For example, in the example of the IEEE 802.11 Wireless Local Area Network Protocol (see Section 6.5.3), for *SIFS*, the constraint $x = 48$ was rescaled to $x = 0.96$, which was then transformed into the constraint $x \geq 0 \land x \leq 1$. This type of overapproximation can result in behaviors which do not correspond to the hypothetical probabilistic timed automata model. As a consequence, computed maximum probabilities on the verified probabilistic timed automata model can be greater than those on the hypothetical probabilistic timed automata model. Vice versa, computed minimum probabilities on the verified probabilistic timed automata model can be less than those on the hypothetical probabilistic timed automata model.

Our approach is different, because we do not introduce intervals in guards to overapproximate rescaled values. For example, we replace the constraint $x = 48$ by the constraint $x = 1$. Indeed, we identify the following problem with the results of [KNS02]: in certain cases, the intervals used included values which do not satisfy K_0. In the case of *SIFS*, for example, the constraint K_0 includes the inequality $\mathit{SIFS} > 0$, and hence it is important that the model cannot behave as if the value of *SIFS* is 0. In the case in which the constraint $x \geq 0 \wedge x \leq 1$ is used, the model can behave as if the value of *SIFS* is 0. Indeed, following the approach of [KNS02], for the verified probabilistic timed automaton with intervals, the computed probabilities were 1, 0.18 and 0.02 for the properties Prop_i for $k = 2, 3$. However, for our PPTA model, for the considered parameter valuations, the computed probabilities were 0.96, 0.06 and 0.002.

Representation of Time. The main reason why PRISM is so sensitive to the size of the constants is because of the use of the digital clock semantics [KNPS06]. PRISM now has a dense-time implementation for probabilistic timed automata which performs better, and which is the subject of ongoing work. This extension of PRISM is based on zone-based probabilistic timed automata model checking techniques [KNP09]. The authors show in particular that forwards reachability techniques can be generalized to produce a stochastic game that yields lower and upper bounds on either minimum or maximum reachability probabilities in probabilistic timed automata. Then, using various refinement methods, the authors are able to tighten these bounds, until they reach the exact value of reachability probabilities in a finite number of steps. The computation time for probabilities using this extension of PRISM is much better than using the standard version of the tool.

Synthesis of Parameters. As noted in [HKM08], parameter synthesis of probabilistic models has received scant attention. Lanotte *et al.* [LMST07] consider parametric discrete-time Markov chains (DTMCs), and establish minimal (and maximal) parameter values for the probabilities associated with transitions in order to ensure reachability properties. Daws [Daw04] also consider DTMCs in which the transition probabilities are parameters. The model checker PARAM [HHWZ10] for parametric DTMCs allows to synthesize symbolic expressions on the parametric probabilities ensuring, e.g., the probability of reaching a goal state. Han *et al.* [HKM08] study continuous-time Markov chains in which the average speed (rate) of state changes are parameters. In contrast to [LMST07, Daw04, HKM08], we consider here the model of probabilistic timed automata. The parameters correspond to the timings that appear in guards of transitions and invariants of locations. Such a parametric framework of probabilistic timed automata appears in [CDF$^+$08], but the model there did not feature nondeterministic choice. In contrast, our model here features *both* nondeterministic and probabilistic choice.

It is also important to point out that, unlike the other chapters of this thesis, we here guarantee not only a qualitative behavior, but also a *quantitative* behavior. Indeed, we guarantee in Chapter 3 the ordering on the actions, but not the global *traversal time* of the system. Similarly, we will introduce in Chapter 7 other methods guaranteeing the shortest path of a graph, but not the *value* of the path itself. In contrast, although we neither guarantee here the value of the timing delays, we give a guarantee on the *value* of the probabilities.

Finally, note that, to our knowledge, no other method allows us to synthesize sets of parameters for which the computation of probabilities is preserved.

Chapter 7

An Inverse Method for Weighted Graphs

– Are you just looking to lose weight, or do you want increased strength and flexibility as well?
– I want to look good naked.

American Beauty
(Sam Mendes)

In Chapter 3, we presented the inverse method in the framework of timed automata, which allows to synthesize a constraint preserving the untimed behavior of the system. In this chapter, we present variants of the inverse method in the frameworks of two kinds of weighted graphs, namely Directed Weighted Graphs, and Markov Decision Processes. The *inverse problem* becomes the following one in this frameworks: "considering the costs of the graph to be unknown constants or *parameters*, given a reference valuation of those costs, compute a constraint on the parameters under which an optimal policy for the reference valuation is still optimal". In the framework of Directed Weighted Graphs, this notion of optimal policy refers to the path of optimal cost between any two points of the graph. In the framework of Markov Decision Processes, it refers to the optimal policy with respect to costs.

As in the framework of timed automata, these variant of the inverse method provide us with some criteria of robustness, in the sense that the initial property is guaranteed around the reference valuation.

We present an application of both methods to simple examples. Two prototype implementations have been done.

Plan of the Chapter. In Section 7.1, we first consider Directed Weighted Graphs. We present the standard formalism, and introduce a parameterization of Directed Weighted Graphs, where costs are seen as parameters. We then state the shortest path guarantee problem, and show how an extension of the Floyd–Warshall algorithm can synthesize constraints on the parameters, guaranteeing that the path of optimal cost in a graph remains the same, for any valuation of the parameters satisfying this constraint. We also present the new prototype tool INSPEQTOR, implementing our algorithm.

We adopt a similar approach in Section 7.2 for Markov Decision Processes. We present the standard formalism, and introduce a parameterization of Markov Decision Processes, where costs are seen as parameters. We then state the optimal policy problem, and introduce an extension of a classical algorithm for policy iteration, which synthesizes constraints on the parameters guaranteeing that the optimal policy in a Markov Decision Process remains the same, for any valuation of the parameters satisfying this constraint. We also present the new prototype tool IMPRATOR, implementing our algorithm.

We finally discuss related work in Section 7.3.

7.1 Directed Weighted Graphs

We are interested in this section in systems modeled by directed weighted graphs. This rather simple structure (obviously less powerful and expressive than timed automata) is made of a set of nodes, and a set of directed edges between some pairs of nodes. Those edges are weighted, i.e., taking a transition has a cost. Directed weighted graphs allow to model various kinds of systems, including timed systems, by considering that the costs associated with the transitions correspond to timing durations. In hardware verification, one may want to model a circuit using a DWG, where the gates are modeled by the nodes of the DWG, and the wires are modeled by the edges. The cost associated with the edge corresponds to the traversal delay of the wires. Note in that case that this model does not allow to consider the functional behavior of the gates (i.e., the logical function associated with the gates).

A classical problem for directed weighted graphs is to compute the optimal path between two given nodes of the graph with respect to the cost associated with the transitions of the path. This optimal path can refer either to the minimum or the maximum cost between those two nodes. In hardware verification, one is generally interested in the *critical path* of an asynchronous circuit, i.e., the longest path between the input and the output of the circuit.

For the sake of simplicity, we will consider in this section the shortest path problem, i.e., the path corresponding to the minimal cost. The results naturally extend to the optimal path problem (viz., shortest or longest path). In the context of timed systems, this corresponds to compute the smallest possible timing delay between two nodes.

7.1.1 The Floyd–Warshall Algorithm

We make use in this section of the standard notations for matrices. In particular, recall that, given a matrix $\mathscr{C}$, we denote by $\mathscr{C}_{ij}$ the element of $\mathscr{C}$ at line i and column j.

We first recall formally the definition of Directed Weighted Graphs. We make use of a standard notation for DWGs, and we will mainly consider that the set Q of nodes is a set of n integers $\{1,\dots,n\}$. We will consider the matrix representation: given a set Q of n nodes, the transitions are represented under the form of a matrix $\mathscr{C}$ of size $n \times n$ indexed by the nodes. For any two nodes i and j, $\mathscr{C}_{ij}$ represents the cost to go from node i to node j. If there is no transition between i and j, then $\mathscr{C}_{ij}$ is set to infinity.

Definition 7.1 (Directed Weighted Graph)**.** A *directed weighted graph (DWG)* $\mathscr{G}$ is a pair $(Q,\mathscr{C})$, where

- Q is a set of nodes, and
- $\mathscr{C}$ is a matrix of costs with values in $\mathbb{R}_{\geq 0} \cup \{\infty\}$ such that, for all $(i,j) \in Q \times Q$, $\mathscr{C}_{ij}$ represents the cost to go from node i to node j; if $\mathscr{C}_{ij} = \infty$, then we consider that there is no transition from i to j.

We assume that, for all $i \in Q$, $\mathscr{C}_{ii} = 0$. ■

The fact that the matrix of costs has its values in $\mathbb{R}_{\geq 0} \cup \{\infty\}$ implies that we do not allow the use of negative costs here. This simplifies subsequently the algorithms and techniques developed in this section. In particular, this allows us to assume that there are no negative cycles (i.e., cycles in the graph whose sum of costs is negative), which allows us to define the Floyd–Warshall algorithm, mentioned below, in a more intuitive way.

We make use of the classical graphical representation for DWGs, where a weighted edge is depicted from node i to node j if $\mathscr{C}_{ij} \neq \infty$.

Example 7.2. Consider the example of DWG depicted in Figure 7.1, containing 6 nodes and 9 weighted transitions. One may see this example as a geographical representation of cities connected with various means of transportation[1]. Then

[1]Actually, this example, taken from [And09b] represents the (real!) shortest path problem between Rennes and LSV in Cachan, using slow train (TER), fast train (TGV), suburbian train (RER), bus or walk. For the reader interested in the practical solution, note that node 1 stands for Rennes, 2 for Le Mans, 3 for Paris, 4 for Massy, 5 for Bagneux and 6 for Cachan.

the cost between two nodes represents the time needed to go from one city to another one.

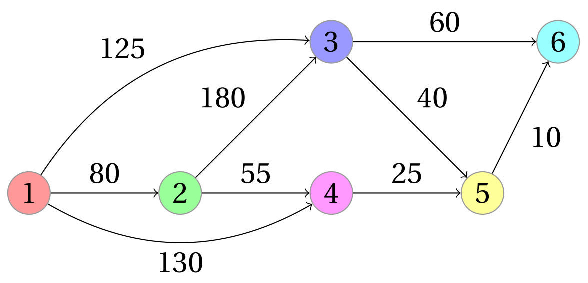


Figure 7.1: An example of Directed Weighted Graph

The corresponding cost matrix $\mathscr{C}$ is the following one:

$$\mathscr{C} = \begin{pmatrix} \infty & 80 & 125 & 130 & \infty & \infty \\ \infty & \infty & 180 & 55 & \infty & \infty \\ \infty & \infty & \infty & \infty & 40 & 60 \\ \infty & \infty & \infty & \infty & 25 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

□

We now recall below the notion of path in the framework of DWGs.

Definition 7.3 (Path)**.** Let $\mathscr{G} = (Q, \mathscr{C})$ be a DWG. A *path* (of length $m-1$) is a sequence of nodes of the form $q_1 \to q_2 \to \cdots \to q_m$, such that $q_i \in Q$ for all $1 \leq i \leq m$ and $\mathscr{C}_{q_i,q_{i+1}} \neq \infty$ for all $1 \leq i \leq m-1$. The node q_{i+1} is called the *successor* of node q_i on this path. The *cost associated with a path* is the sum of the costs of any two successors of the path, i.e., $\sum_{i=1}^{m} \mathscr{C}_{q_i,q_{i+1}}$. ■

We depict paths under a graphical form using boxed nodes and double arrows labeled with costs.

Example 7.4. Consider again the DWG of Example 7.2. We give in Figure 7.2 an example of path of length 3 for this DWG, going from node 1 to node 6.

$$\boxed{1} \xRightarrow{130} \boxed{4} \xRightarrow{25} \boxed{5} \xRightarrow{10} \boxed{6}$$

Figure 7.2: Example of path of a DWG

The cost associated with this path is 165.

□

We now define the notion of shortest path. Recall that we consider here the notion of path of *minimal* cost. This work naturally extends to the optimal path (i.e., path of minimal or maximal cost).

Definition 7.5 (Shortest path)**.** Let $\mathscr{G}$ be a DWG, and i and j two nodes of $\mathscr{G}$. A path P from i and j is said to be a *shortest path* from i to j if no other path from i to j has an associated cost strictly smaller than the cost associated with P. ■

Note that, given two nodes i and j of a DWG, several paths can be defined as a shortest path from i and j (if their associated cost is the same).

Let us consider the *shortest path problem*, consisting in finding a path of minimal cost between any two nodes of a DWG.

A solution to the shortest path problem is given by the Floyd–Warshall algorithm [Flo62], which computes the shortest path between any pairs of nodes of a DWG. This algorithm $FW(\mathscr{G})$, recalled in Algorithm 7, computes on the one hand the shortest path matrix $\mathcal{V}$, giving the cost of the shortest path between any two nodes of the graph, and computes on the other hand the successor matrix $\mathscr{S}$, such that $\mathscr{S}_{ij}$ gives the successor node of a node i on a shortest path to j, for any pair of nodes (i, j). We see matrices as two-dimensional arrays that one can *update* using operator $\leftarrow$. The value ∞ is defined as follows: for all $i \in \mathbb{R}_{\geq 0}$, $i + \infty = \infty$, $\min(i, \infty) = i$ and we have $i \leq \infty$ if and only if $i \neq \infty$. This last equivalence means in particular that the relation $\infty \leq \infty$ does not hold.

We initialize matrix $\mathscr{S}$ to $\mathbf{0}$, which is a matrix of same size as $\mathscr{S}$ where all values are set to $\emptyset$, where $\emptyset$ is the notation for an undefined node.

Algorithm 7: Floyd–Warshall algorithm $FW(\mathscr{G})$

```
input  : A DWG $\mathscr{G} = (Q, \mathscr{C})$
output: $\mathcal{V}$: shortest path matrix
output: $\mathscr{S}$: successor matrix

1 $\mathcal{V} \leftarrow \mathscr{C}$
2 $\mathscr{S} \leftarrow \mathbf{0}$
3 for $k = 1$ to $n$ do
4     for $i = 1$ to $n$ do
5         for $j = 1$ to $n$ do
6             $\mathcal{V}_{ij} \leftarrow \min(\mathcal{V}_{ij}, \mathcal{V}_{ik} + \mathcal{V}_{kj})$
7             $\mathscr{S}_{ij} \leftarrow \begin{cases} j & \textbf{if } \mathcal{V}_{ij} \leq \mathcal{V}_{ik} + \mathcal{V}_{kj} \\ k & \textbf{if } \mathcal{V}_{ik} + \mathcal{V}_{kj} < \mathcal{V}_{ij} \end{cases}$
```

Example 7.6. Let us apply the Floyd–Warshall algorithm to the DWG of Example 7.2. Initially, we have:

$$\mathcal{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & \infty & \infty \\ \infty & \infty & 180 & 55 & \infty & \infty \\ \infty & \infty & \infty & \infty & 40 & 60 \\ \infty & \infty & \infty & \infty & 25 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \qquad \mathscr{S} = \begin{pmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

Let us consider the first iteration of the outer loop (with $k = 1$). For $i = 1$ and $j = 2$, we have that $\mathcal{V}_{12} \le \mathcal{V}_{11} + \mathcal{V}_{12}$. Hence, we update $\mathscr{S}$ as follows: $\mathscr{S}_{12} \leftarrow 2$. Similarly, for $i = 1$ and $j = 3$, we have that $\mathcal{V}_{13} \le \mathcal{V}_{11} + \mathcal{V}_{13}$. Hence, we update $\mathscr{S}$ as follows: $\mathscr{S}_{13} \leftarrow 3$. We continue this process with the other entries of $\mathscr{S}$. At the end of this first iteration, matrix $\mathcal{V}$ remains unchanged, and $\mathscr{S}$ has been updated several times. We have:

$$\mathcal{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & \infty & \infty \\ \infty & \infty & 180 & 55 & \infty & \infty \\ \infty & \infty & \infty & \infty & 40 & 60 \\ \infty & \infty & \infty & \infty & 25 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \qquad \mathscr{S} = \begin{pmatrix} \emptyset & 2 & 3 & 4 & \emptyset & \emptyset \\ \emptyset & \emptyset & 3 & 4 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

The next iteration of the outer loop (for $k = 2$) leaves both matrices $\mathcal{V}$ and $\mathscr{S}$ unchanged. This comes from the fact that the direct path from node 1 to nodes 3 and 4 is better (i.e., of lower cost) than through node 2.

Now consider the third iteration of the outer loop (for $k = 3$). For $i = 1$ and $j = 5$, we have that $\mathcal{V}_{13} + \mathcal{V}_{35} \le \mathcal{V}_{15}$, because $\mathcal{V}_{15}$ is infinite. Hence, we update $\mathscr{S}_{15} \leftarrow 3$, $\mathcal{V}_{15} \leftarrow \mathcal{V}_{13} + \mathcal{V}_{35} = 165$. Similarly, for $i = 1$ and $j = 6$, we update $\mathscr{S}_{16} \leftarrow 3$, $\mathcal{V}_{16} \leftarrow \mathcal{V}_{13} + \mathcal{V}_{36} = 185$. Similarly for $i = 2$ and $j = 5$, and for $i = 2$ and $j = 6$. This gives:

$$\mathcal{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & 165 & 185 \\ \infty & \infty & 180 & 55 & 220 & 240 \\ \infty & \infty & \infty & \infty & 40 & 60 \\ \infty & \infty & \infty & \infty & 25 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \qquad \mathscr{S} = \begin{pmatrix} \emptyset & 2 & 3 & 4 & 3 & 3 \\ \emptyset & \emptyset & 3 & 4 & 3 & 3 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

For $k = 4$, it turns out that node 5 has a lower cost from node 1 and 2 through node 4 rather than through node 3. We now have:

$$\mathscr{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & 155 & 185 \\ \infty & \infty & 180 & 55 & 80 & 240 \\ \infty & \infty & \infty & \infty & 40 & 60 \\ \infty & \infty & \infty & \infty & 25 & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \qquad \mathscr{S} = \begin{pmatrix} \emptyset & 2 & 3 & 4 & 4 & 3 \\ \emptyset & \emptyset & 3 & 4 & 4 & 3 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

For $k = 5$, it turns out that node 6 is reachable from nodes 1, 2, 3 and 4 through node 5, which always has a lower cost than through the current node. For $k = 6$, the matrices remains unchanged. Hence, at the end of the algorithm, we have:

$$\mathscr{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & 155 & 165 \\ \infty & \infty & 180 & 55 & 80 & 90 \\ \infty & \infty & \infty & \infty & 40 & 50 \\ \infty & \infty & \infty & \infty & 25 & 35 \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \qquad \mathscr{S} = \begin{pmatrix} \emptyset & 2 & 3 & 4 & 4 & 5 \\ \emptyset & \emptyset & 3 & 4 & 4 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

□

Observe that, although the *cost* of the shortest path between any two nodes of the DWG is computed by the Floyd–Warshall algorithm (this is matrix $\mathscr{V}$), the shortest *path* itself is not explicitly computed. However, it can be retrieved in a very straightforward manner using $\mathscr{S}$, which gives the successor of the node on the shortest path to another node.

Example 7.7. Consider again the DWG of Example 7.2. Let us compute the shortest path from node 1 to node 6. Using the matrices $\mathscr{V}$ and $\mathscr{S}$ computed by the Floyd–Warshall algorithm (see Example 7.6 above), one sees that the cost of the shortest path from node 1 to node 6 is $\mathscr{V}_{1,6} = 165$. From matrix $\mathscr{S}$, one sees that a successor of 1 on the way to 6 is $\mathscr{S}_{1,6} = 5$. Recursively, a successor of 1 on the way to 5 is $\mathscr{S}_{1,5} = 4$. Since the successor of 1 on the way to 4 is 4 and the successor of 5 on the way to 6 is 6, the shortest path from node 1 to node 6 is $1 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6$. This corresponds actually to the path considered in Example 7.4. □

7.1.2 The Problem

Now suppose that some of the costs of a DWG are not reliable, i.e., can be subject to changes. One may wonder whether the shortest path between any two nodes (computed, e.g., using the Floyd–Warshall algorithm) remains the same. More generally, we are interested to know until which augmentation or diminution of the costs the shortest paths remain the same.

This approach can be motivated in the framework of timed systems, where costs actually model timing delays, to verify if the change of a delay will impact the shortest paths between any two nodes. More specifically, in hardware verification, this is of interest to know whether the replacement of a gate or a wire by another one (faster or slower) will or not impact the global behavior of the system, and in particular the shortest (or longest) path between the input and the output.

A solution to this problem would be to change the delays, and apply again the Floyd–Warshall algorithm. However, this may be time-consuming if several delays are subject to changes. Moreover, when looking for the *maximal* (or minimal) possible change of a particular delay without impacting the shortest path, several applications of the Floyd–Warshall algorithm will only result in an *approximate* value for the optimal delay (e.g., using a dichotomic approach), because this optimal delay may be rational valued. As a consequence, as in the framework of Timed Automata considered earlier in this thesis, it is interesting to reason *parametrically,* and synthesize a *constraint* on the costs seen as parameters guaranteeing that the shortest paths remain the same.

Parametric Graphs. We now introduce the formalism of *parametric* directed weighted graphs, i.e., directed weighted graphs where the costs are not any longer constants but *parameters.* Recall that a parameter is a constant with *unknown value.* This parametrization of DWGs into *parametric* directed weighted graphs is actually similar to the parametrization of TAs into PTAs (see Section 2.2).

The following definition introduces more formally Parametric Directed Weighted Graphs [And09b]. [2]

Definition 7.8 (Parametric Directed Weighted Graph)**.** A *Parametric Directed Weighted Graph (PDWG)* $\mathcal{G}$ is a triple $(Q, P, \mathcal{D})$, where:

- Q is a set of nodes,
- P is a set of parameters, and

[2]The name of Parametric Weighted Graphs may refer in the literature to graphs with a single parameter x, where costs are functions (e.g., linear or polynomial) of x. Our formalism is different but, in order to remain consistent with the names used in this thesis (e.g., Parametric Timed Automata or Parametric Probabilistic Timed Automata), we keep the name of Parametric Directed Weighted Graphs.

- $\mathscr{D}$ is a matrix of costs with values in $P \cup \{\infty\}$ such that, for all $(i, j) \in Q \times Q$, $\mathscr{D}_{ij}$ represents the parametric cost to go from node i to node j; if $\mathscr{D}_{ij} = \infty$, we consider that there is no transition from i to j.

We assume that, for all $i \in Q$, $\mathscr{C}_{ii} = 0$. ■

Note that one could easily extend this formalism to costs with values in $P \cup \mathbb{R} \cup \{\infty\}$, i.e., allowing the use of both parametric or constant costs. However, note that parameters have values in $\mathbb{R}_{\geq 0}$, not in $\mathbb{R}_{\geq 0} \cup \{\infty\}$: this means that a parametric cost cannot be infinite, and thus that the transition to which it is associated does exist with a (finite) cost, whatever the value of the parameter is.

We represent PDWGs under a graphical form in the same way as for DWGs, except that the costs associated with the edges are no longer constants but parameters.

Example 7.9. Consider the example of PDWG depicted in Figure 7.3, containing 6 nodes, 9 weighted transitions and 9 parameters.[3]

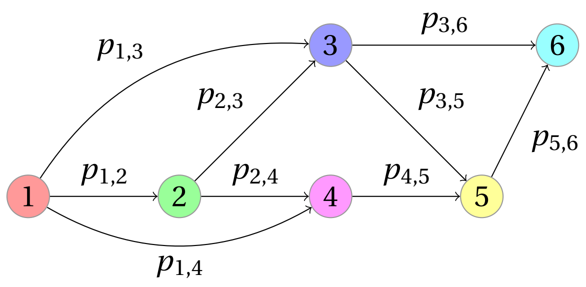


Figure 7.3: An example of PDWG

The corresponding parametric cost matrix $\mathscr{D}$ is the following one:

$$\mathscr{D} = \begin{pmatrix} \infty & p_{1,2} & p_{1,3} & p_{1,4} & \infty & \infty \\ \infty & \infty & p_{2,3} & p_{2,4} & \infty & \infty \\ \infty & \infty & \infty & \infty & p_{3,5} & p_{3,6} \\ \infty & \infty & \infty & \infty & p_{4,5} & \infty \\ \infty & \infty & \infty & \infty & \infty & p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

□

[3]For the sake of simplicity, we do not give to our parameters names of the form p_i, but of the form $p_{i,j}$ to underline the fact that this parameter corresponds to the cost associated with the edge from node i to node j.

Let $\mathscr{G} = (Q, P, \mathscr{D})$ be a PDWG. Given a valuation $\pi = \{\pi_1, \dots, \pi_M\}$ of the parameters, we denote by $\mathscr{D}[\pi]$ the matrix of costs where each parameter p_i has been replaced with value π_i. By extension, we denote by $\mathscr{G}[\pi]$ the standard non-parametric DWG $(Q, \mathscr{D}[\pi])$.

Example 7.10. Consider again the PDWG, say $\mathscr{G}$, of Example 7.9. Consider also the following valuation π_0 of the parameters:

$p_{1,2} = 80$	$p_{1,3} = 125$	$p_{1,4} = 130$
$p_{2,3} = 180$	$p_{2,4} = 55$	$p_{3,5} = 40$
$p_{3,6} = 60$	$p_{4,5} = 25$	$p_{5,6} = 10$

Then the standard, non-parametric DWG $\mathscr{G}[\pi_0]$ is the DWG considered in Example 7.2. $\square$

Recall that we now suppose that some of the costs of a DWG are not reliable, i.e., can be subject to changes. The problem we are interested in is the following one: "until which variation of the costs of a DWG seen as parameters the shortest paths between any two nodes of the DWG remain the same?"

More formally, we state below the problem we face in this section.

The shortest path guarantee problem
Let $\mathscr{G}$ be a PDWG, and π_0 a reference valuation of the parameters. Synthesize a constraint K_0 on the parameters such that:

1. $\pi_0 \models K_0$, and
2. for all $\pi \models K_0$, given any two nodes i and j of $\mathscr{G}$, a shortest path between i and j in $\mathscr{G}[\pi_0]$ is also a shortest path between i and j in $\mathscr{G}[\pi]$.

7.1.3 An Inverse Method for Weighted Graphs

Let us adapt the Floyd–Warshall algorithm (given in Algorithm 7) in order to synthesize a constraint K_0 solving the shortest path guarantee problem.

We will follow in a straightforward manner the inverse method algorithm *IM* defined in the framework of PTAs (see Algorithm 1 in Chapter 3). As a consequence, we will synthesize a constraint at each modification of $\mathscr{S}_{ij}$. This constraint will be of the form $\mathscr{W}_{ik} + \mathscr{W}_{kj} \leq \mathscr{W}_{ij}$ or $\mathscr{W}_{ij} \leq \mathscr{W}_{ik} + \mathscr{W}_{kj}$. At the end of the algorithm, those constraints guarantee that, for each parameter valuation modeling this constraint, each shortest path in the original DWG is also a shortest path in the DWG instantiated with this parameter valuation.

This algorithm, denoted by *PFW*($\mathscr{G}$), is given in Algorithm 8 [And09b]. It takes as an input a PDWG $\mathscr{G} = (Q, P, \mathscr{D})$ and a valuation π_0 of the parameters, and returns a constraint K_0 on the parameters solving the shortest path guarantee problem.

Algorithm 8: Parametric Floyd–Warshall algorithm $PFW(\mathscr{G}, \pi_0)$

```
input    : A DWG $\mathscr{G} = (Q, P, \mathscr{D})$
input    : A reference valuation $\pi_0$
output   : $K_0$: constraint on the parameters
variable: $\mathscr{V}$: shortest path matrix
variable: $\mathscr{W}$: parametric shortest path matrix
variable: $\mathscr{S}$: successor matrix
1  $\mathscr{V} \leftarrow \mathscr{D}[\pi_0]$
2  $\mathscr{W} \leftarrow \mathscr{D}$
3  $\mathscr{S} \leftarrow \mathbf{0}$
4  $K_0 \leftarrow$ true
5  for $k = 1$ to $n$ do
6      for $i = 1$ to $n$ do
7          for $j = 1$ to $n$ do
8              if $\mathscr{V}_{ij} \leq \mathscr{V}_{ik} + \mathscr{V}_{kj}$ then
9                  $\mathscr{S}_{ij} \leftarrow j$;
10                 $K_0 \leftarrow K_0 \wedge \{\mathscr{W}_{ij} \leq \mathscr{W}_{ik} + \mathscr{W}_{kj}\}$;
11             else if $\mathscr{V}_{ik} + \mathscr{V}_{kj} < \mathscr{V}_{ij}$ then
12                 $\mathscr{S}_{ij} \leftarrow k$;
13                 $K_0 \leftarrow K_0 \wedge \{\mathscr{W}_{ik} + \mathscr{W}_{kj} \leq \mathscr{W}_{ij}\}$;
14                 $\mathscr{V}_{ij} \leftarrow \mathscr{V}_{ik} + \mathscr{V}_{kj}$;
15                 $\mathscr{W}_{ij} \leftarrow \mathscr{W}_{ik} + \mathscr{W}_{kj}$;
```

This set of nested loops computes, as in the standard Floyd–Warshall algorithm, the (instantiated) shortest path matrix $\mathscr{V}$, as well as the successor matrix $\mathscr{S}$. Note that this latter matrix is not used in the computation of K_0, and is only given for the sake of consistency with the standard Floyd–Warshall algorithm. Moreover, the algorithm also computes the matrix $\mathscr{W}$ of the *parametric* shortest path. Each element of $\mathscr{W}_{ij}$ contains the sum of parametric costs (i.e., parameters) corresponding to the shortest path from i to j, or is equal to ∞ if there is no path from i to j. Note that, by construction, we have that $\mathscr{V} = \mathscr{W}[\pi_0]$.

When a modification of the successor matrix $\mathscr{S}$ is done (i.e., in both the **if** and the **else if** conditions), we synthesize a new inequality within K_0. The addition of an inequality in the **if** condition (resp. in the **else if** condition) can be summarized as follows: in the shortest path from i to j, the path through k is necessarily longer (resp. shorter) or equal to the path directly from i to j. As a consequence, this constraint guarantees that the shortest path is indeed the one computed by the classical Floyd–Warshall algorithm.

Note that the addition of an inequality to K_0 has been simplified in Algorithm 8 for the sake of understanding. No inequality is actually synthesized if one of the two members is infinite, i.e., if there is no path from i to j, from i to k, or from k to j. Moreover, we only synthesize a large inequality in the case where the path through k is strictly smaller than the path through j (i.e., in the **else if** condition) because we are interested in guaranteeing that a shortest path under π_0 remains *one* shortest path under π, not necessarily the only one (see Remark 7.15 page 186).

Example 7.11. Consider again the PDWG, say $\mathscr{G}$, of Example 7.9. Consider also the following valuation π_0 of the parameters:

$$\begin{array}{lll} p_{1,2} = 80 & p_{1,3} = 125 & p_{1,4} = 130 \\ p_{2,3} = 180 & p_{2,4} = 55 & p_{3,5} = 40 \\ p_{3,6} = 60 & p_{4,5} = 25 & p_{5,6} = 10 \end{array}$$

Let us apply Algorithm *PFW* to $\mathscr{G}$ and π_0. We will not explicitly recall the computation of $\mathscr{V}$ and $\mathscr{S}$, which is exactly the same as in Example 7.6, and focus only on the computation of $\mathscr{W}$, and of the set K_0 of inequalities. Initially, we have:

$$\mathscr{W} = \begin{pmatrix} \infty & p_{1,2} & p_{1,3} & p_{1,4} & \infty & \infty \\ \infty & \infty & p_{2,3} & p_{2,4} & \infty & \infty \\ \infty & \infty & \infty & \infty & p_{3,5} & p_{3,6} \\ \infty & \infty & \infty & \infty & p_{4,5} & \infty \\ \infty & \infty & \infty & \infty & \infty & p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Let us consider the first iteration of the outer loop (with $k = 1$). For $i = 1$ and $j = 2$, we have that $\mathscr{V}_{12} \leq \mathscr{V}_{11} + \mathscr{V}_{12}$. Hence, as in Example 7.6, we update $\mathscr{S}$ as follows: $\mathscr{S}_{12} \leftarrow 2$. However, we do not synthesize the inequality $\mathscr{W}_{12} \leq \mathscr{W}_{11} + \mathscr{W}_{12}$ because $\mathscr{W}_{11}$ is infinite. Similarly, for $i = 1$ and $j = 3$, we have that $\mathscr{V}_{13} \leq \mathscr{V}_{11} + \mathscr{V}_{13}$. Hence, we update $\mathscr{S}$ as follows: $\mathscr{S}_{13} \leftarrow 3$. Again we do not synthesize any inequality because $\mathscr{W}_{11}$ is infinite. The first iteration of the outer loop continues in this manner. At the end of this first iteration, only $\mathscr{S}$ has been updated (see Example 7.6), and matrices $\mathscr{V}$ and $\mathscr{W}$ remain unchanged.

Now consider the second iteration of the outer loop (for $k = 2$). For $i = 1$ and $j = 3$, we have that $\mathscr{V}_{13} \leq \mathscr{V}_{12} + \mathscr{V}_{23}$. We update $\mathscr{S}$ as follows: $\mathscr{S}_{13} \leftarrow 3$ (which actually leaves the matrix unchanged). We synthesize the inequality $\mathscr{W}_{13} \leq \mathscr{W}_{12} + \mathscr{W}_{23}$, which gives after replacement:

$$p_{1,3} \leq p_{1,2} + p_{2,3}$$

Similarly, for $i = 1$ and $j = 4$, we have that $\mathscr{V}_{14} \leq \mathscr{V}_{12} + \mathscr{V}_{24}$. We update $\mathscr{S}$ as follows: $\mathscr{S}_{14} \leftarrow 4$ (which actually leaves the matrix unchanged). We synthesize the inequality $\mathscr{W}_{14} \leq \mathscr{W}_{12} + \mathscr{W}_{24}$, which gives after replacement:

$$p_{1,4} \leq p_{1,2} + p_{2,4}$$

Now consider the third iteration of the outer loop (for $k = 3$). For $i = 1$ and $j = 5$, we have that $\mathscr{V}_{13} + \mathscr{V}_{35} \leq \mathscr{V}_{15}$, because $\mathscr{V}_{15}$ is infinite. Hence, we update $\mathscr{S}_{15} \leftarrow 3$, $\mathscr{V}_{15} \leftarrow \mathscr{V}_{13} + \mathscr{V}_{35} = 165$, $\mathscr{W}_{15} \leftarrow \mathscr{W}_{13} + \mathscr{W}_{35}$, and we do not synthesize any inequality, because $\mathscr{W}_{15}$ is infinite. Similarly, for $i = 1$ and $j = 6$, we update $\mathscr{S}_{16} \leftarrow 3$, $\mathscr{V}_{16} \leftarrow \mathscr{V}_{13} + \mathscr{V}_{36} = 185$, $\mathscr{W}_{16} \leftarrow \mathscr{W}_{13} + \mathscr{W}_{36}$, and we do not synthesize any inequality, because $\mathscr{W}_{16}$ is infinite. Similarly for $i = 2$ and $j = 5$, and for $i = 2$ and $j = 6$. We now have:

$$\mathscr{W} = \begin{pmatrix} \infty & p_{1,2} & p_{1,3} & p_{1,4} & p_{1,3} + p_{3,5} & p_{1,3} + p_{3,6} \\ \infty & \infty & p_{2,3} & p_{2,4} & p_{2,3} + p_{3,5} & p_{2,3} + p_{3,6} \\ \infty & \infty & \infty & \infty & p_{3,5} & p_{3,6} \\ \infty & \infty & \infty & \infty & p_{4,5} & \infty \\ \infty & \infty & \infty & \infty & \infty & p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Now consider the fourth iteration of the outer loop (for $k = 4$). For $i = 1$ and $j = 5$, we have that $\mathscr{V}_{14} + \mathscr{V}_{45} \leq \mathscr{V}_{15}$. Hence, we update $\mathscr{S}_{15} \leftarrow 4$, $\mathscr{V}_{15} \leftarrow \mathscr{V}_{14} + \mathscr{V}_{45} = 155$, $\mathscr{W}_{15} \leftarrow \mathscr{W}_{14} + \mathscr{W}_{45}$, and we synthesize the inequality $\mathscr{W}_{14} + \mathscr{W}_{45} \leq \mathscr{W}_{15}$, which gives after replacement:

$$p_{1,4} + p_{1,5} \leq p_{1,3} + p_{3,5}$$

For $i = 2$ and $j = 5$, we have similarly that $\mathscr{V}_{24} + \mathscr{V}_{45} \leq \mathscr{V}_{25}$. Hence, we update $\mathscr{S}_{25} \leftarrow 4$, $\mathscr{V}_{25} \leftarrow \mathscr{V}_{24} + \mathscr{V}_{45} = 80$, $\mathscr{W}_{25} \leftarrow \mathscr{W}_{24} + \mathscr{W}_{45}$, and we synthesize the inequality $\mathscr{W}_{24} + \mathscr{W}_{45} \leq \mathscr{W}_{25}$, which gives after replacement:

$$p_{2,4} + p_{4,5} \leq p_{2,3} + p_{3,5}$$

We now have:

$$\mathscr{W} = \begin{pmatrix} \infty & p_{1,2} & p_{1,3} & p_{1,4} & p_{1,4} + p_{4,5} & p_{1,3} + p_{3,6} \\ \infty & \infty & p_{2,3} & p_{2,4} & p_{2,4} + p_{4,5} & p_{2,3} + p_{3,6} \\ \infty & \infty & \infty & \infty & p_{3,5} & p_{3,6} \\ \infty & \infty & \infty & \infty & p_{4,5} & \infty \\ \infty & \infty & \infty & \infty & \infty & p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

We give with less details the fifth iteration of the outer loop (for $k = 5$), and only mention the inequalities synthesized. For $i = 1$ and $j = 6$, we synthesize the inequality $\mathscr{W}_{15} + \mathscr{W}_{56} \leq \mathscr{W}_{16}$, which gives after replacement:

$$p_{1,4} + p_{4,5} + p_{5,6} \leq p_{1,3} + p_{3,6}$$

For $i = 2$ and $j = 6$, we synthesize the inequality $\mathscr{W}_{25} + \mathscr{W}_{56} \leq \mathscr{W}_{26}$, which gives after replacement:

$$p_{2,4} + p_{4,5} + p_{5,6} \leq p_{2,3} + p_{3,6}$$

For $i = 3$ and $j = 6$, we synthesize the inequality $\mathscr{W}_{35} + \mathscr{W}_{56} \leq \mathscr{W}_{36}$, which gives after replacement:

$$p_{3,5} + p_{5,6} \leq p_{3,6}$$

For $k = 6$, all matrices remain unchanged, no inequality is synthesized, and the algorithm terminates. We finally have:

$$\mathscr{W} = \begin{pmatrix} \infty & p_{1,2} & p_{1,3} & p_{1,4} & p_{1,4}+p_{4,5} & p_{1,4}+p_{4,5}+p_{5,6} \\ \infty & \infty & p_{2,3} & p_{2,4} & p_{2,4}+p_{4,5} & p_{2,4}+p_{4,5}+p_{5,6} \\ \infty & \infty & \infty & \infty & p_{3,5} & p_{3,5}+p_{5,6} \\ \infty & \infty & \infty & \infty & p_{4,5} & p_{4,5}+p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & p_{5,6} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Also recall $\mathscr{V}$ and $\mathscr{S}$:

$$\mathscr{V} = \begin{pmatrix} \infty & 80 & 125 & 130 & 155 & 165 \\ \infty & \infty & 180 & 55 & 80 & 90 \\ \infty & \infty & \infty & \infty & 40 & 50 \\ \infty & \infty & \infty & \infty & 25 & 35 \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \quad \mathscr{S} = \begin{pmatrix} \emptyset & 2 & 3 & 4 & 4 & 5 \\ \emptyset & \emptyset & 3 & 4 & 4 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & 5 & 5 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & 6 \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

This gives us the following final constraint K_0:

$$\begin{array}{rr} & p_{1,3} \leq p_{1,2} + p_{2,3} \\ \wedge & p_{1,4} \leq p_{1,2} + p_{2,4} \\ \wedge & p_{1,4} + p_{4,5} \leq p_{1,3} + p_{3,5} \\ \wedge & p_{2,4} + p_{4,5} \leq p_{2,3} + p_{3,5} \\ \wedge & p_{3,5} + p_{5,6} \leq p_{3,6} \\ \wedge & p_{2,4} + p_{4,5} + p_{5,6} \leq p_{2,3} + p_{3,6} \\ \wedge & p_{1,4} + p_{4,5} + p_{5,6} \leq p_{1,3} + p_{3,6} \end{array}$$

Now, as an example of application, suppose that one wants to maximize, say, $p_{4,5}$ without changing any of the shortest paths of the system. One can instantiate all the parameters except $p_{4,5}$ in the constraint K_0, which gives:

$$p_{4,5} \leq 35$$

Then, for $p_{4,5}$ up to 35, the shortest paths of the system all remain the same. □

7.1.4 Properties

It is easy to see that the algorithm *PFW* always terminates.

Lemma 7.12 (Termination)**.** *Let $\mathcal{G}$ be a PDWG, and π_0 a reference valuation of the parameters. Then the algorithm $K_0 = PFW(\mathcal{G}, \pi_0)$ terminates.*

Proof. From the finiteness of the number of iterations of the loops. ■

Similarly to Lemma 3.2 in the framework of the inverse method for PTAs, we now show that $\pi_0 \models K_0$.

Lemma 7.13. *Let $\mathcal{G}$ be a PDWG, and π_0 a reference valuation of the parameters. Let $K_0 = PFW(\mathcal{G}, \pi_0)$. Then $\pi_0 \models K_0$.*

Proof. All inequalities in K_0 are either of the form $\mathcal{W}_{ij} \geq \mathcal{W}_{ik} + \mathcal{W}_{kj}$, or $\mathcal{W}_{ij} \leq \mathcal{W}_{ik} + \mathcal{W}_{kj}$, for some i, j and k.

1. First consider the case an inequality J of the form $\mathcal{W}_{ij} \geq \mathcal{W}_{ik} + \mathcal{W}_{kj}$. Due to the construction of this inequality in our algorithm, this inequality has been synthesized in the **if** condition where $\mathcal{V}_{ij} \geq \mathcal{V}_{ik} + \mathcal{V}_{kj}$. Since we have by construction that $\mathcal{V} = \mathcal{W}[\pi_0]$, then this inequality is satisfiable by π_0.

2. Now consider the case an inequality J of the form $\mathcal{W}_{ij} \leq \mathcal{W}_{ik} + \mathcal{W}_{kj}$. Similarly, due to the construction of this inequality in our algorithm, this inequality has been synthesized in the **if** condition where $\mathcal{V}_{ij} \leq \mathcal{V}_{ik} + \mathcal{V}_{kj}$. Then this inequality is satisfiable by π_0.

■

We can now state that our algorithm solves the shortest path guarantee problem.

Proposition 7.14 (Correctness)**.** *Let $\mathcal{G}$ be a PDWG, and π_0 a reference valuation of the parameters. Let $K_0 = PFW(\mathcal{G}, \pi_0)$. Then*

1. *$\pi_0 \models K_0$, and*

2. *for all $\pi \models K_0$, given any two nodes i and j of $\mathcal{G}$, a shortest path between i and j in $\mathcal{G}[\pi_0]$ is also a shortest path between i and j in $\mathcal{G}[\pi]$.*

Proof. Item 1 is proved by Lemma 7.13. Let us now show by *reductio ad absurdum* that for all $\pi \models K_0$, given any two nodes i and j of $\mathcal{G}$, a shortest path between i and j in $\mathcal{G}[\pi_0]$ is also a shortest path between i and j in $\mathcal{G}[\pi]$.

Let i and j be two nodes of $\mathcal{G}$. Consider a shortest path between i and j in $\mathcal{G}[\pi_0]$ of the form $i \to k \to \cdots \to j$, which is not a shortest path between i

and j in $\mathcal{G}[\pi]$. As a consequence, there exists a strictly shorter path between i and j in $\mathcal{G}[\pi]$. Suppose this path is of the form $i \to l \to \cdots \to j$. However, at some point of the algorithm, we had that $\mathcal{V}_{ik} + \mathcal{V}_{kj} \leq \mathcal{V}_{il} + \mathcal{V}_{lj}$, because k is a better node than l on the path from i to j in $\mathcal{G}[\pi_0]$. So, according to the algorithm, we synthesized an inequality of the form $\mathcal{W}_{ik} + \mathcal{W}_{kj} \leq \mathcal{W}_{il} + \mathcal{W}_{lj}$. Instantiated by π, this would give that k is a better node than l on the path from i to j in $\mathcal{G}[\pi]$, which is not possible because $i \to k \to \cdots \to j$ is not a shortest path between i and j in $\mathcal{G}[\pi]$. ■

When considering the complexity for the algorithm, it is clear that the number of comparisons of our algorithm *PFW* is exactly the same as in the standard Floyd–Warshall algorithm, i.e., is in $O(n^3)$, where n is the number of nodes of the PDWG. The upper bound on the number of synthesized inequalities of Algorithm *PFW* is n^3.

Remark 7.15. In this form, the algorithm only guarantees that a shortest path in $\mathcal{G}[\pi_0]$ remains a shortest path in $\mathcal{G}[\pi]$, for any $\pi \models \mathit{PFW}(\mathcal{G}, \pi_0)$. However, the reciprocal statement is not true: a shortest path in $\mathcal{G}[\pi]$ is not necessarily a shortest path in $\mathcal{G}[\pi_0]$. In order to have this statement satisfied, one should modify the algorithm as follows: depending on the strict or large inequality in the **if** comparison, one should synthesize large or strict inequalities. □

7.1.5 Implementation and Case Studies

This algorithm *PFW* has been implemented in a prototype named INSPEQTOR (for *INference of Shortest Paths with EQuivalent Time-abstract behaviOR*), a program written in OCaml, containing about 3000 lines of code.

By applying this tool INSPEQTOR to the PDWG of Figure 7.3 modeling Example 7.2, the constraint K_0 given previously was synthesized in less than 0.01 second (experiment performed on an Intel Quad Core 3 GHz with 3.2 Gb memory).

Discussion. This method is of interest for the study of *weighted* systems, with costs that can correspond to any kind of variable (money, temperature, distance, etc.). In particular, the method can be applied to some classes of timed systems; in that case, the costs correspond to timing delays. In the case of hardware verification, one can model components of a circuit with nodes of a DWG, and timing delays associated with wires with edges. Other representations can be chosen, when the delays associated with wires are negligible but the delays associated with the traversal of components are not. It may be of interest to know if it is possible to change a component (or wire) of the system by another component (or wire) of different delay, without changing the overall behavior of the system. By synthesizing a constraint using our algorithm, one can guarantee that the shortest paths of the system will not change.

In the case of hardware verification, note that this algorithm *PFW* is adapted to the study of components where the functionality of the components is not taken into account. Indeed, the model of DWGs may be used to compute an approximation of the response time of the circuit. If one wants to take into account the functionalities of the elements (AND gates, OR gates, etc.) but there is no memory element, the time separation of events is the most accurate model; recall that the inverse method was initially defined in this framework [EF08]. When one wants to take into account the functionalities of the elements and in the case where there are memory elements, as well as unbounded loops, the model of Timed Automata is the most accurate model, to which one can then apply the inverse method as defined in Chapter 3.

7.2 Markov Decision Processes

We are interested in this section in systems modeled by Markov Decision Processes. This extension of weighted directed graphs is made of a set of states, with labeled probabilistic transitions. When in a given state, one can *choose* nondeterministically a given action. Then, for this state and this action, it is possible to reach several destination states with some probability. The sum of the probabilities associated with the transitions leaving a given state through a given action is equal to one. To each of those probabilistic transitions, we associate a weight (or cost).

One can associate to a Markov Decision Process a *policy*, i.e., a function which associates to every state a single action. This solves the nondeterminism in the Markov Decision Process, which then behaves like a Markov Chain [KMST59]. A classical problem for Markov Decision Processes consists in finding the *optimal* policy with respect to the weights of the system. The weight of a *path* (or sequence of transitions) is the sum of the weights of its constitutive actions. The *value* (or cost) of a given policy v for a given state s corresponds to the mean weight of the paths induced by v, which go from s to a final state of the graph.

We solve in this section the following inverse problem for Markov Decision Processes: "Given a Markov Decision Process and an optimal policy, find values for the weights seen as parameters such that, for any valuation of the parameters, the optimal policy remain optimal".

7.2.1 Preliminaries

Markov Decision Processes are widely used to model systems involving probabilities and costs together, e.g., the power consumption of devices (see, e.g., [PBBDM98]).

We associate to every edge of the graph a *probability* such that, for a given state and a given action, the sum of the probabilities of the edges leaving this state through this action is equal to 1. We recall below the formal definition of Markov Decision Processes.

Definition 7.16 (Markov Decision Process)**.** A *Markov Decision Process (MDP)* is a tuple $\mathcal{M} = (Q, \Sigma, Prob, w)$, where

- $Q = \{s_1, \dots, s_n\}$ is a set of states;
- Σ is a set of actions;
- $Prob : Q \times \Sigma \times Q \to [0,1]$ is a probability function such that $Prob(s_1, a, s_2)$ is the probability that action a in state s_1 will lead to state s_2, and $\forall s \in Q, \forall a \in \Sigma : \sum_{s' \in Q} Prob(s, a, s') = 1$;
- $w : Q \times \Sigma \to \mathbb{R}$ is a weight function such that $w(s,a)$ (also denoted by $w_a(s)$) is the weight associated with the action a when leaving s.

■

As for DWGs, we make here use of the term “weight” in order to denote the value associated with a transition. Note that the terms of “cost” or “reward” are also used in the literature. Note also that, in order to remain consistent with the literature, *nodes* of DWGs are called *states* in the framework of MDPs.

We follow the usual conventions for the graphical representation of MDPs: states are represented by nodes; probabilistic edges are represented by arcs from states, labeled by the action name and its associated weight, and which split into multiple arcs, each of which leads to a state and is labeled by a probability. As for Probabilistic Timed Automata (see Chapter 6), probabilistic edges which correspond to probability 1 are illustrated by a single arc from state to state, labeled only by the action name and its associated weight.

Example 7.17. We give in Figure 7.4 an example of Markov Decision Process with 4 states (viz., s_1, s_2, s_3 and s_4) and 4 actions (viz., a, b, c and d).

When in state s_1, one can choose either action a or action b. When choosing action a, one can go either to state s_1 (with probability 0.3) or to state s_2 (with probability 0.7), both with a weight of 5. When choosing action b, one can go either to state s_2 (with probability 0.5) or to state s_3 (with probability 0.5), both with a weight of 2. The rest of this MDP can be explained similarly. Note that

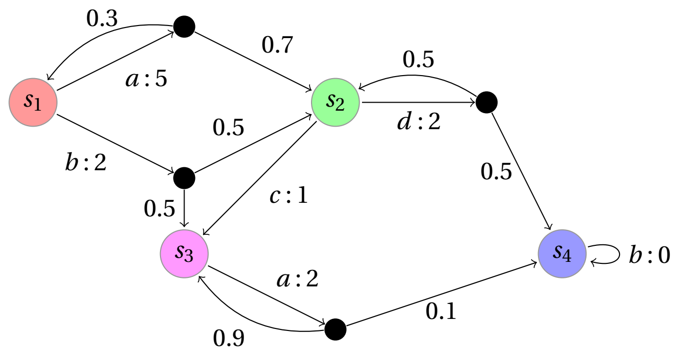


Figure 7.4: An example of Markov Decision Process

the transition from state s_2 to state s_3 (labeled by action c and weight 1) has probability 1, and similarly for the self loop in state s_4 (labeled by action b and weight 0). □

Note that, for the sake of simplicity, we consider in Definition 7.16 that the weight function takes as input one state and one action only. This implies that our MDPs are such that, for a given state s and a given action a, the weight corresponding to a transition from s to a destination state s' through action a is the same for any destination state s'; only the probability differs. In the case where one would need to express different weights for different transitions starting from the same state through the same action, it is possible to modify the model in a straightforward manner.

Absorbing state. In the following, we consider the MDP $\mathcal{M} = (Q, \Sigma, Prob, w)$. Given a state $s \in Q$, we denote by $enabled(s)$ the set of possible actions for s, i.e., $\{a \in \Sigma \mid \exists s' \in Q : Prob(s, a, s') > 0\}$.

Example 7.18. For the MDP of Example 7.17, we have:

$$\begin{array}{rcl} enabled(s_1) & = & \{a, b\} \\ enabled(s_2) & = & \{c, d\} \\ enabled(s_3) & = & \{a\} \\ enabled(s_4) & = & \{b\} \end{array}$$

□

We suppose that, for any state $s \in Q$, $enabled(s) \neq \emptyset$. We also suppose that $\mathcal{M}$ has a *unique* "absorbing state", i.e., a state which is reachable (with positive probability) from any other state for any policy (see definition below), and which has a self-loop outgoing transition with weight 0 and probability 1. [4] We suppose in the following that the absorbing state is s_n.

Example 7.19. In the MDP of Example 7.17, s_4 is the unique absorbing state. □

Policy. In every state s of $Q \setminus \{s_n\}$, we can choose *nondeterministically* an action a in $\mathit{enabled}(s)$. Then, for this action, the system will evolve to a state s' such that $\mathit{Prob}(s, a, s') > 0$. A way of removing nondeterminism from an MDP is to introduce a (deterministic, state-based) *policy* v, i.e., a function from states to actions. A policy is of the form $v = \{s_1 \to a_{i_1}, s_2 \to a_{i_2}, \ldots, s_n \to a_{i_n}\}$, with $a_{i_1}, \ldots, a_{i_n} \in \Sigma$. Note that we usually do not associate a policy to the absorbing state s_n. The following definition formally defines the notion of policy.

The literature often makes use of letter π for a policy. However, in this thesis, π always corresponds to a valuation of the parameters. As a consequence, we will use in the following letter v to denote a policy.

Definition 7.20 (Policy)**.** Let $\mathscr{M} = (Q, \Sigma, \mathit{Prob}, w)$ be an MDP. A *policy* for $\mathscr{M}$ is a total function $v : Q \to \Sigma$ such that $\forall s \in Q : v[s] \in \mathit{enabled}(s)$. ■

By this definition, we make the assumption that, given an MDP $\mathscr{M}$, a policy v and a state s, we have that $\mathit{Prob}(s, v[s], s') > 0$ for some $s' \in Q$.

Given an MDP $\mathscr{M}$, a policy v and a state s, observe that we chose to denote by $v[s]$ the action associated with state s. We use the notation $v[s]$ (and not, e.g., $v(s)$) because we will consider in our algorithms v to be a vector (or array) indexed by the states.

It can be shown that an MDP associated with a policy behaves as a *Markov chain* [KMST59], because of the removal of nondeterminism.

Example 7.21. Consider again Example 7.17. We associate to this MDP the following policy: $v = \{s_1 \to a,\ s_2 \to d,\ s_3 \to a\}$. Then the MDP of Figure 7.4 associated with v is depicted in Figure 7.5.

□

Value determination. Given a policy v, the *associated value* is a function mapping each state s to the mean sum of weights attached to the paths induced by v, which go from s to the absorbing state s_n. (By convention, the value associated with s_n is null.)

[4] Given an MDP with no absorbing state and whose graph is strongly connected, it is possible to turn it into an MDP containing an absorbing state, by adding a *discount* d (typically close to 1, e.g., $d = 0.99$). In that case, we multiply all existing probabilities of the MDP by the discount d, and, for each state s and for each action $a \in \mathit{enabled}(s)$, we add a transition from s by a to the absorbing state with probability $1 - d$. Thus, we consider the probabilistic behavior of the system within $(1 - d)^{-1}$ units of time.

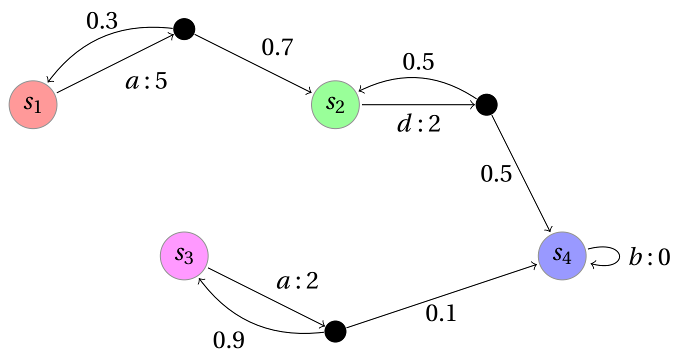


Figure 7.5: Our example of MDP associated with policy ν

This value function can be computed using the value determination algorithm *VD*($\mathcal{M}, \nu$), that we recall in Algorithm 9. Given an MDP $\mathcal{M}$ and a policy ν, this algorithm computes the value V introduced above, i.e., the mean sum of weights attached to the paths reaching s_n, for each starting state in $Q \setminus \{s_n\}$. We denote by $v[s]$ the value associated with state s.

Algorithm 9: Algorithm of value determination *VD*($\mathcal{M}, \nu$)

```
input  : $\mathcal{M}$: Markov Decision Process $(Q, \Sigma, Prob, w)$
input  : $\nu$: Policy
output: $v$: value function
1 SOLVE
```

$$\{v[s] = w_{\nu[s]}(s) + \sum_{s' \in Q} Prob(s, \nu[s], s') \times v[s']\}_{s \in Q \setminus s_n}$$

The value v computed by Algorithm *VD* is obtained by solving a system of linear equations, and is computed using operations on matrices and vectors. The fact that there is a single solution to this system is due to the fact that the matrix *Prob* restricted with ν is invertible, which comes itself from the existence of an absorbing state.

Example 7.22. Consider again the MDP of Example 7.17, and the policy $\nu = \{s_1 \to a,\ s_2 \to d,\ s_3 \to a\}$. Then the value function can be computed by solving the following system:

$$\begin{cases} v[s_1] &= w_a(s_1) + \sum_{s' \in Q} Prob(s_1, a, s') \times v[s'] \\ v[s_2] &= w_d(s_2) + \sum_{s' \in Q} Prob(s_2, d, s') \times v[s'] \\ v[s_3] &= w_a(s_3) + \sum_{s' \in Q} Prob(s_3, a, s') \times v[s'] \end{cases}$$

By replacing the weights and probabilities by their value, this system becomes:

$$\begin{cases} v[s_1] &= 5 + 0.3 * v[s_1] + 0.7 * v[s_2] \\ v[s_2] &= 2 + 0.5 * v[s_2] + 0.5 * v[s_4] \\ v[s_3] &= 2 + 0.9 * v[s_3] + 0.1 * v[s_4] \end{cases}$$

By solving this system, one finds out that the value function is:

$$\begin{cases} v[s_1] &= \frac{78}{7} \\ v[s_2] &= 4 \\ v[s_3] &= 20 \end{cases}$$

□

Optimal policy. A classical problem for MDPs is to find an *optimal policy*, i.e., a policy under which the value function is optimal, for every $s \in Q$. Note that, under the assumption of the existence of an absorbing state, such an optimal policy always exists, but is not necessarily unique (see, e.g., [How60]). We focus here on finding an optimal policy for which the value function is *minimal*.

Definition 7.23 (Optimal policy)**.** Let $\mathcal{M} = (Q, \Sigma, Prob, w)$ be an MDP, and ν a policy of $\mathcal{M}$. We say that ν is *optimal* for $\mathcal{M}$ if, for each state $s \in Q$:

$$\nu(s) \quad = \quad \underset{a}{\arg\min} \left\{ \sum_{s'} Prob(s, \nu[s], s') \left(w_a(s) + v[s'] \right) \right\}$$

■

Various algorithms have been proposed in order to compute an optimal policy of an MDP. These algorithms usually make use of two arrays (or vectors) indexed by the states: the array of values v, which associates a value to each state, and the policy itself, which associates an action to each state. We will consider here one of the most widely used, viz., the policy iteration algorithm.

The policy iteration algorithm. Given an MDP $\mathcal{M}$, the policy iteration algorithm computes an optimal policy for $\mathcal{M}$. We present the classical policy iteration algorithm $PI(\mathcal{M})$ in Algorithm 10 [How60]. In order to compute the optimal policy for an MDP, this algorithm makes use of the algorithm *VD* for value determination in MDPs.

The algorithm starts with an arbitrary policy, and consists in a loop repeated until fixpoint. One first computes the value (using Algorithm *VD*) associated

Algorithm 10: Algorithm of policy iteration *PI*($\mathscr{M}$)

```
input  : $\mathscr{M}$: Markov Decision Process $(Q, \Sigma, Prob, w)$
output: $\nu$: Policy optimal w.r.t. $w$ (initially arbitrary)
output: $v$: Value function
repeat
    $v \leftarrow VD(\mathscr{M}, \nu)$;
    $fixpoint \leftarrow$ true
    foreach $s \in Q \setminus s_n$ do
        $optimum \leftarrow v[s]$
        foreach $a \in enabled(s)$ do
            if $w_a(s) + \sum_{s' \in Q} Prob(s, a, s') v[s'] < optimum$ then
                $optimum \leftarrow w_a(s) + \sum_{s' \in Q} Prob(s, a, s') v[s']$;
                $\nu[s] \leftarrow a$;
                $fixpoint \leftarrow$ false ;
until $fixpoint$;
```

with the MDP under the current policy. Then, for each state s of the MDP (except the absorbing state), for each action a enabled for this state s, one computes the sum of the weight associated with s and a with the sum of the values of the destination states of s through a weighted by their respective probabilities. If this value is strictly less than the current optimal value for s (i.e., the value associated with the current policy for s), then it means than action a is strictly better than the current policy $\nu[s]$, and the policy is updated in consequence. If the policy is not changed for any state, then the fixpoint is reached, and both the optimal policy and the associated value are returned.

Example 7.24. Let us apply the policy iteration algorithm to the MDP of Example 7.17. Let us start from the arbitrary policy $\nu_0 = \{s_1 \to a,\ \ s_2 \to c,\ \ s_3 \to a\}$. The initial value function is $v_0 = \{s_1 \to \frac{197}{7},\ s_2 \to 21,\ s_3 \to 20\}$.

At the first iteration of the external loop of Algorithm *PI*, it turns out that b is a strictly better policy for s_1 than a, and that d is a strictly better policy for s_2 than c. Then, the new policy becomes $\nu_1 = \{s_1 \to a,\ \ s_2 \to c,\ \ s_3 \to a\}$. The new value function is $v_1 = \{s_1 \to 14,\ s_2 \to 4,\ s_3 \to 20\}$.

At the second iteration of Algorithm *PI*, it turns out that a is a strictly better policy for s_1 than b. Then, the new policy becomes $\nu_2 = \{s_1 \to a,\ \ s_2 \to d,\ \ s_3 \to a\}$. The new value function is $v_2 = \{s_1 \to \frac{78}{7},\ s_2 \to 4,\ s_3 \to 20\}$.

At the third iteration, the policy cannot be improved. Then ν_2 is the optimal policy for this MDP. □

7.2.2 The Inverse Problem for MDPs

Let us consider again the MDP of Example 7.17. Suppose that some weight of the MDP changed; for example, let us assume that the weight $w_d(s_2)$ of leaving state s_2 through action d is not 2 anymore, but now 3. The question that arises is: "does the optimal policy computed previously remain optimal?" One may apply again the policy iteration algorithm to this modified MDP, and check that the optimal policy remains the same (which would actually be the case). Now, let us assume that $w_d(s_2) = 4$. One should then again apply the policy iteration algorithm, and so on for any value. More generally, we are interested in the following question: "what is the maximal possible value for $w_d(s_2)$ such that the optimal policy v remains optimal?" In other terms, we want to be able to maximize or minimize some of the weights without changing the optimal policy.

We state this problem more formally below [AF09].

The optimal policy guarantee problem
Let $\mathscr{M}$ be a MDP, and v an optimal policy. Synthesize a constraint K_0 on the weights of $\mathscr{M}$ seen as parameters such that:

1. $\pi_0 \models K_0$, and
2. for all $\pi \models K_0$, v remains an optimal policy for $\mathscr{M}[\pi]$.

7.2.3 An Inverse Method for MDPs

We first adapt the notion of MDP to the parametric case [AF09]. We now consider that the weights of the MDP are *parameters*, i.e., unknown constants. This parametrization of MDPs is actually similar to the parametrization of TAs into PTAs (see Chapter 2). Note also that, as in Chapter 6, we do *not* parameterize the values of the probabilities, but the values of the constants involved in the transitions (here, the weights).

Definition 7.25. A *Parametric Markov Decision Process (PMDP)* is a tuple $\mathscr{M} = (Q, \Sigma, P, Prob, W)$, where

- $Q = \{s_1, \dots, s_n\}$ is a set of states;
- Σ is a set of actions;
- P is a set of parameters;
- $Prob : Q \times \Sigma \times Q \to [0,1]$ is a probability function such that $Prob(s_1, a, s_2)$ is the probability that action a in state s_1 will lead to state s_2, and $\forall s \in Q, \forall a \in \Sigma : \sum_{s' \in Q} Prob(s, a, s') = 1$;

- $W : Q \times \Sigma \to P$ is a parametric weight function such that $W(s,a)$ (also denoted by $W_a(s)$) is a parameter associated with the action a when leaving s.

■

We consider in the following the PMDP $\mathcal{M} = (Q, \Sigma, P, Prob, W)$. Given a valuation π of the parameters, we denote by $W[\pi]$ the function from $Q \times \Sigma$ to $\mathbb{R}$ obtained by replacing each occurrence of a parameter p_i in W with the value $\pi(p_i)$, for $1 \leq i \leq N$. By extension, we denote by $\mathcal{M}[\pi]$ the (standard) MDP $(Q, \Sigma, Prob, W[\pi])$.

Example 7.26. We give in Figure 7.6 an example of Parametric Markov Decision Process $\mathcal{M}$ with 4 states (viz., s_1, s_2, s_3 and s_4), 4 actions (viz., a, b, c and d), and 5 parameters (viz., p_{1a}, p_{1b}, p_{2c}, p_{2d} and p_{3a}). Note that, for the sake of simplicity, we do not give to our parameters names of the form p_i, but of the form p_{sa} to underline the fact that this parameter corresponds to the weight of leaving state s through action a.

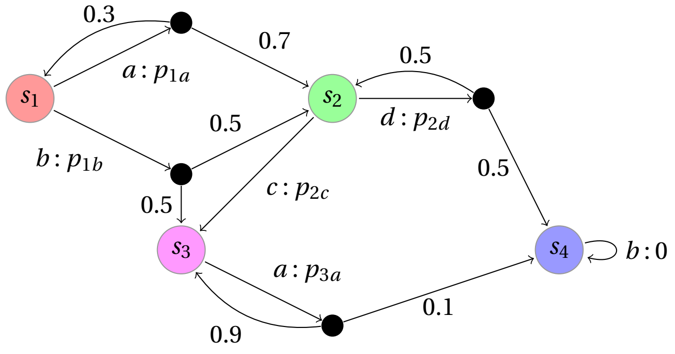


Figure 7.6: An example of Parametric Markov Decision Process

When in state s_1, one can choose either action a or action b. When choosing action a, one can go either to state s_1 (with probability 0.3) or to state s_2 (with probability 0.7), both with a parametric weight equal to p_{1a}. When choosing action b, one can go either to state s_2 (with probability 0.5) or to state s_3 (with probability 0.5), both with a parametric weight equal to p_{1b}. The rest of this MDP can be explained similarly.

Consider the following valuation of the parameters: $\pi_0 : p_{1a} = 5 \wedge p_{1b} = 2 \wedge p_{2c} = 1 \wedge p_{2d} = 2 \wedge p_{3a} = 2$. Then $\mathcal{M}[\pi_0]$ corresponds to the (non-parametric) MDP of Example 7.17. □

We will solve the inverse problem for MDPs by introducing a parametric form of the algorithm *PI*. We first need to compute the *parametric* value associated with every state s of a PMDP for a given policy ν, i.e., the mean sum of the parametric weights associated with the paths induced by v going from s to s_n. We introduce the algorithm *pVD*, which computes, given a policy v, the parametric value associated with the states of the PMDP. This algorithm is a straightforward adaptation to the parametric case of the classical algorithm *VD* of value determination for MDPs (see Algorithm 9 page 191). We denote by $V[s]$ the parametric value associated with state s.

This algorithm *pVD*($\mathcal{M}, v$) is given in Algorithm 11 [AF09].

Algorithm 11: Parametric value determination *pVD*($\mathcal{M}, v$)

```
input : $\mathcal{M}$: Parametric Markov Decision Process $(Q, \Sigma, P, Prob, W)$
input : $v$: Policy
output: $V$: Parametric value function
1 SOLVE
      $\{V[s] = W_{v[s]}(s) + \sum_{s' \in Q} Prob(s, v[s], s') \times V[s']\}_{s \in Q \setminus \{s_n\}}$
```

The value V computed by this algorithm *pVD* is obtained by solving a system of parametric linear equations. This system can be seen as an equation on vectors and matrices of the form $V = A \times V + B$. As a consequence, this system is equivalent to $V = (1 - A)^{-1} \times B$, and can be implemented using the inversion of matrix $(1 - A)$. Note that this matrix A is computed from matrix *Prob* and vector v, and is therefore a constant real-valued matrix (i.e., containing no parameters). Only B is a parametric vector. As for the algorithm *VD*, the fact that there is a single solution to this system comes from the existence of an absorbing state. Note also that the parametric value associated with a state is a *linear term*, as defined in Definition 2.1.

Example 7.27. Consider again the PMDP of Example 7.26, and the policy $v = \{s_1 \to a,\ s_2 \to d,\ s_3 \to a\}$. Then the parametric value function can be computed by solving the following system:

$$\left\{\begin{array}{lcl} V[s_1] & = & W_a(s_1) + \sum_{s' \in Q} Prob(s_1, a, s') \times V[s'] \\ V[s_2] & = & W_d(s_2) + \sum_{s' \in Q} Prob(s_2, d, s') \times V[s'] \\ V[s_3] & = & W_a(s_3) + \sum_{s' \in Q} Prob(s_3, a, s') \times V[s'] \end{array}\right.$$

By replacing the parametric weights and the probabilities by their value, this system becomes:

$$\left\{\begin{array}{lcl} V[s_1] & = & p_{1a} + 0.3 * V[s_1] + 0.7 * V[s_2] \\ V[s_2] & = & p_{2d} + 0.5 * V[s_2] + 0.5 * V[s_4] \\ V[s_3] & = & p_{3a} + 0.9 * V[s_3] + 0.1 * V[s_4] \end{array}\right.$$

By solving this system, one finds out that the parametric value function is:

$$\begin{cases} V[s_1] & = & \frac{10}{7}p_{1a} + 2p_{2d} \\ V[s_2] & = & 2p_{2d} \\ V[s_3] & = & 10p_{3a} \end{cases}$$

□

We state in the following lemma that, given a PMDP $\mathcal{M}$ and a policy v, the instantiation with π of the parametric value associated with $\mathcal{M}$ w.r.t. v is equal to the value associated with $\mathcal{M}[\pi]$ w.r.t. v. We use $V[\pi]$ to denote the standard (non-parametric) value function equal to V in which each parameter has been instantiated with its value π.

Lemma 7.28. *Let $\mathcal{M}$ be a PMDP, π a valuation of the parameters, and v a policy for $\mathcal{M}$. Let $V = pVD(\mathcal{M}, v)$. Then $V[\pi] = VD(\mathcal{M}[\pi], v)$.*

Proof. Let $\mathcal{M} = (Q, \Sigma, P, Prob, W)$ be a PMDP. The algorithm *pVD*$(\mathcal{M}, v)$ consists in solving a system of the form $V = A \times V + W_v$, where W_v denotes the matrix W restricted to policy v. Hence, $V = (1 - A)^{-1} \times W_v$. Moreover, the algorithm *VD*$(\mathcal{M}[\pi], v)$ consists in solving a system of the form $v = A' \times v + W[\pi]_v$, i.e., $v = (1 - A')^{-1} \times W[\pi]_v$. It is easy to see from the two algorithms that $A = A'$. We trivially have: for all s, $W[\pi]_{v[s]}(s) = (W_{v[s]}(s))[\pi]$, where $(W_{v[s]}(s))[\pi]$ denotes the linear term $W_{v[s]}(s)$ where every occurrence of a parameter p_i was replaced by its instantiation π_i. Hence, $V[\pi] = VD(\mathcal{M}[\pi], v)$. ■

We now introduce the algorithm *pPI*, which solves the problem stated in Section 7.2.2. Given a reference valuation π_0 of the parameters, this algorithm takes as input a PMDP $\mathcal{M}$, and an optimal policy v_0 associated with $\mathcal{M}[\pi_0]$ (which can be computed using *PI*$(\mathcal{M}[\pi_0])$). Recall that, by "optimal", we mean here a policy under which the value of states is *minimal*. The algorithm outputs a constraint K_0 on the parameters such that:

1. $\pi_0 \models K_0$, and
2. for any $\pi \models K_0$, v_0 is an optimal policy of $\mathcal{M}[\pi]$.

The algorithm *pPI* is given in Algorithm 12 [AF09]. We can summarize this algorithm as follows:

1. Compute the parametric value function, which associates to any state a parametric value w.r.t. v_0, using Algorithm *pVD*;

```
Algorithm 12: Inverse method algorithm for MDPs $pPI(\mathcal{M}, \pi_0, v_0)$
  input  : $\mathcal{M}$: Parametric Markov Decision Process $(Q, \Sigma, P, Prob, W)$
  input  : $v$: Policy
  output: $K_0$: Constraint on the set of parameters
1 $V \leftarrow pVD(\mathcal{M}, v_0)$;
2 $K_0 \leftarrow$ true;
3 foreach $s \in Q \setminus \{s_n\}$ do
4     foreach $a \in enabled(s)$ s.t. $a \neq v_0[s]$ do
5         $K_0 := K_0 \wedge \{W_a(s) + \sum_{s' \in Q} Prob(s, a, s') V[s'] \geq V[s]\}$
```

2. For each state $s \neq s_n$, for each action a different from the action $v_0[s]$ given by the optimal policy, synthesize the following inequality stating that a is not a better action (i.e., an action which would lead to a better policy) than $v_0[s]$:

$$W_a(s) + \sum_{s' \in Q} Prob(s, a, s') V[s'] \geq V[s]$$

The above set of inequalities implies that, for any s and a, the policy obtained from v_0 by changing $v_0[s]$ with a, does not improve policy v_0 (i.e., does not lead to any smaller value of state).

Example 7.29. Consider again the PMDP $\mathcal{M}$ of Example 7.26, and the policy $v_0 = \{s_1 \to a, \; s_2 \to d, \; s_3 \to a\}$. Consider the following valuation π_0 of the parameters: $\pi_0 : p_{1a} = 5 \wedge p_{1b} = 2 \wedge p_{2c} = 1 \wedge p_{2d} = 2 \wedge p_{3a} = 2$.

Recall from Example 7.24 that v_0 is an optimal policy for $\mathcal{M}[\pi_0]$.

The computation of $pVD(\mathcal{M}, v_0)$ (see Example 7.27) gives:

$$\begin{cases} V[s_1] &=& \frac{10}{7} p_{1a} + 2p_{2d} \\ V[s_2] &=& 2p_{2d} \\ V[s_3] &=& 10p_{3a} \end{cases}$$

Let us apply our algorithm *pPI* to $\mathcal{M}$ and v_0. First recall from Example 7.18 that the *enabled* function is defined as follows:

$$\begin{array}{rcl} enabled(s_1) &=& \{a, b\} \\ enabled(s_2) &=& \{c, d\} \\ enabled(s_3) &=& \{a\} \\ enabled(s_4) &=& \{b\} \end{array}$$

Now, we can synthesize the following inequalities by applying algorithm *pPI*.

$$\begin{array}{rrcl} & p_{1b} + \frac{1}{2} V(s_2) + \frac{1}{2} V(s_3) &\geq& V(s_1) \\ \wedge & p_{2c} + V(s_3) &\geq& V(s_2) \end{array}$$

The first inequality is synthesized for state s_1 and action b, whereas the second one is synthesized for state s_2 and action c. After replacement of V by the parametric value computed above, one gets:

$$\begin{array}{rrcl} & p_{1b}+5p_{3a} & \geq & \frac{10}{7}p_{1a}+p_{2d} \\ \wedge & p_{2c}+10p_{3a} & \geq & 2p_{2d} \end{array}$$

□

7.2.4 Properties

We first state that Algorithm *pPI* terminates.

Proposition 7.30 (Termination)**.** *Let $\mathcal{M}$ be a PMDP, and v_0 a policy. Then algorithm pPI$(\mathcal{M}, v_0)$ terminates.*

Proof. Since $\mathcal{M}$ contains exactly one absorbing state, the computation of the parametric value in *pVD* is guaranteed to terminate with a single solution. Since the number of synthesized inequalities is finite, it is easy to see that Algorithm *pPI* terminates. ■

We now show that, given a reference valuation π_0 of the parameters, if v_0 is an optimal policy for $\mathcal{M}[\pi_0]$, then π_0 models the constraint K_0 output by our algorithm. This proof is similar the proof of Lemma 7.13.

Lemma 7.31. *Let $\mathcal{M}$ be a PMDP, π_0 a valuation of the parameters, and v_0 an optimal policy of $\mathcal{M}[\pi_0]$. Let $K_0 = pPI(\mathcal{M}, v_0)$. Then $\pi_0 \models K_0$.*

Proof. (By *reductio ad absurdum*) Suppose $\pi_0 \not\models K_0$. Then, there exists an inequality J in K_0 such that $\pi_0 \not\models J$. By construction, this inequality J is of the form $W_a(s) + \sum_{s' \in Q} Prob(s, a, s')V[s'] \geq V[s]$, for some s and some a. If this inequality J is not satisfied by π_0, this means that a is a strictly better policy for s than the policy $v_0[s]$ in $\mathcal{M}[\pi_0]$, which is not possible since v_0 is an optimal policy for $\mathcal{M}[\pi_0]$. ■

We now state that our algorithm *pPI* solves the inverse problem as described in Section 7.2.2.

Proposition 7.32 (Correctness)**.** *Let $\mathcal{M}$ be a PMDP, π_0 a valuation of the parameters, and v_0 an optimal policy of $\mathcal{M}[\pi_0]$. Let $K_0 = pPI(\mathcal{M}, v_0)$. Then:*

1. *$\pi_0 \models K_0$, and*
2. *for all $\pi \models K_0$, the policy v_0 is optimal for $\mathcal{M}[\pi]$.*

Proof. The proof of item (1) comes from Lemma 7.31. Let us prove item (2) by *reductio ad absurdum.* Recall that $\mathcal{M} = (Q, \Sigma, P, Prob, W)$. Let $\pi \models K_0$. We have $\mathcal{M}[\pi] = (Q, \Sigma, Prob, W[\pi])$.

Suppose that v_0 is not an optimal policy for $\mathcal{M}[\pi]$. Let v be an optimal policy for $\mathcal{M}[\pi]$. Then there exists some state s such that $v[s]$ is a strictly better policy than $v_0[s]$ for $\mathcal{M}[\pi]$. Let $a = v[s]$ and $a_0 = v_0[s]$. Let $\upsilon = VD(\mathcal{M}[\pi], v)$. Since a is a strictly better policy than a_0 for state s in $\mathcal{M}[\pi]$, then, from the last iteration of Algorithm $PI(\mathcal{M}[\pi])$, we have: $W[\pi]_{a_0}(s) + \sum_{s' \in Q} Prob(s, a_0, s')\,\upsilon[s'] > W[\pi]_a(s) + \sum_{s' \in Q} Prob(s, a, s')\,\upsilon[s']$.

Moreover, since $a \neq v_0[s]$, Algorithm $pPI(\mathcal{M}, v_0)$ synthesizes the following inequality in K_0: $W_a(s) + \sum_{s' \in Q} Prob(s, a, s') V[s'] \geq V[s]$. Since $V[s] = W_{a_0}(s) + \sum_{s' \in Q} Prob(s, a_0, s') \times V[s']$ (from the call to Algorithm $pVD(\mathcal{M}, v_0)$), this inequality is equal to $W_a(s) + \sum_{s' \in Q} Prob(s, a, s') V[s'] \geq W_{a_0}(s) + \sum_{s' \in Q} Prob(s, a_0, s') \times V[s']$. Since $\pi \models K_0$, the instantiation of K_0, and in particular of this inequality, with π should evaluate to true. By Lemma 7.28, we have $V[\pi] = \upsilon$. Hence, by instantiating the inequality with π, we get: $W[\pi]_a(s) + \sum_{s' \in Q} Prob(s, a, s')\,\upsilon[s'] \geq W[\pi]_{a_0}(s) + \sum_{s' \in Q} Prob(s, a_0, s') \times \upsilon[s']$, which is exactly the contrary of what was stated before. ■

When considering the complexity, first observe that the size (in term of number of inequalities) of the constraint K_0 output by our algorithm is in $O(|Q| * |\Sigma|)$.

Moreover, our algorithm *pPI* has a smaller worst case complexity than the standard algorithm *PI* for policy iteration. First note that Algorithm *pVD* is a straightforward adaptation of *VD*, and thus has the same complexity as *VD*. Whereas *PI* may call *VD* up to $|Q|^{|\Sigma|}$ times in the worst case (where $|Q|$ and $|\Sigma|$ denote the number of states and actions of the MDP respectively), our algorithm *pPI* calls *pVD* only once. Moreover, whereas *PI* may perform up to $|Q|^{|\Sigma|} * |Q| * |\Sigma|$ comparisons (line 7 of Algorithm 10), the generation of the constraints of our algorithm is only in $O(|Q| * |\Sigma|)$. As a consequence, once one knows one optimal policy for a PMDP for a given valuation of the parameters, it is then cheap to compute a constraint using our algorithm in order to find more values of the parameters with the same policy.

Example 7.33. We can now answer to the question from Section 7.2.2 regarding our example of PMDP described in Example 7.26, which was: "what is the maximal possible value for $w_d(s_2)$ such that the optimal policy v remains optimal?" By instantiating all parameters except p_{2d} within the constraint synthesized in Example 7.29, one gets the inequality $p_{2d} \leq \frac{34}{7}$. As a consequence, $\frac{34}{7}$ represents the maximal possible value for $w_d(s_2)$ such that the optimal policy v remains optimal, and solves our problem. □

7.2.5 Implementation and Applications

The algorithm *pPI* has been implemented under the form of a program named IMPRATOR (standing for *Inverse Method for Policy with Reward AbstracT behaviOR*). This program, containing about 4300 lines of code, is written in OCaml, and uses matrix inversion to compute the parametric value V in Algorithm *pVD*. Note that IMPRATOR also allows the use of MDPs with no absorbing state, by adding a *discount* (see footnote page 190 for more details).

We applied our program to various examples of MDPs modeling devices. For a system containing 11 states, 4 actions and 132 transitions, corresponding to the model of a robot evolving in a bounded physical space [SB03], our program IMPRATOR synthesizes a constraint in 0.17 s.

The program and various case studies can be downloaded on the IMPRATOR Web page[5].

Finally note that we are studying the adaptation of the method to Markov decision processes with *two* weights, as used in the problem of *dynamic power management* [PBBDM98] for real-time systems where one wants to minimize the power consumption while keeping a certain level of efficiency.

7.3 Related Work

The authors of [ROC05] address the same problem as the problem we faced in Section 7.1. Indeed, they determine the maximum and minimum weights that each edge can have so that a given path remains optimal. They also show that their technique is in $O(m+K\,log K)$ time, where m is the number of edges in the graph, and K is the number of edges on the given optimal path. A difference with our work is that they infer the minimum and maximum *constant* weights, whereas by reasoning parametrically we are able to infer relations (under the form of a constraint) between the weights of the graph.

In [CFLY10], the authors consider *parametric weighted graphs* as graphs whose edges are labeled with continuous real function of a *single* variable. As in our framework, the instantiation of such a parametric weighted graphs results in a classical directed weighted graph. The authors of [CFLY10] introduce two-phase algorithms for weighted graphs. The first phase takes as input a parametric weighted graph, and synthesizes a data structure which will be used later for the second phase, viz., the computation of the shortest path for a particular instantiation of the parameter. The main interest of this work relies in the fact that one needs to perform only once the first phase, and the complexity of the second phase is smaller than the usual complexity to solve the shortest path

[5]`http://www.lsv.ens-cachan.fr/~andre/ImPrator/`

problem on a given (instantiated) weighted graph. Our approach differs mainly on two points. First, we use a different parameter on each edge of the graph, whereas they use a (linear or polynomial) function on a single variable. Second, one can see the work of [CFLY10] as a *direct problem*, because the authors aim at solving the shortest path problem for a single valuation of the parameter using a parametric structure, whereas our inverse method aims at synthesizing a constraint on the parameters using a reference instantiation.

When considering parametrized versions of probabilistic models, recall that Lanotte *et al.* [LMST07] consider parametric discrete-time Markov chains (DTMCs), and establish minimal and maximal parameter values for the probabilities associated with transitions in order to ensure reachability properties.

Finally note that we also introduced in [AF09] another inverse method on DWGs allowing to synthesize a constraint on the weights seen as parameters guaranteeing that the circuit of maximal mean of a DWG remains the same. Recall that a *circuit of maximal mean* is a circuit of the graph with the highest sum of weights divided by the number of nodes of the circuit. We proceeded in a similar way as in Section 7.1 and Section 7.2, by extending to the parametric case an algorithm defined in [CTCG$^+$98], and computing the circuit of maximal mean of a DWG.

Chapter 8

Conclusion and Perspectives

– I'll be back before you can say blueberry pie.
– Blueberry pie.
– Maybe not that fast, but pretty fast, okay?

Pulp Fiction
(Quentin Tarantino)

8.1 Summary

We proposed in this thesis methods for the synthesis of delays for timed systems. In Chapter 3, we introduced the inverse method algorithm in the framework of timed automata, allowing to synthesize a constraint on the delays viewed as parameters, guaranteeing the same time-abstract behavior as for the reference valuation. In particular, this time-abstract equivalence preserves linear-time properties. This allows a notion of robustness: one can guarantee that the values *around* a given value of the delays will not impact the time-abstract overall behavior of the system. Moreover, it allows the optimization of some of the timing delays, without changing the overall behavior of the system.

In Chapter 4, we presented our implementation, IMITATOR II, allowing us to synthesize constraints guaranteeing the good behavior of various case studies, hardware devices and communication protocols. In particular, we were able to verify a portion of the SPSMALL memory designed by the chipset manufacturer ST-Microelectronics, and relax timing bounds of the input signals.

By iterating this inverse method on various points of a bounded parameter domain, we showed in Chapter 5 how to partition the parametric space into good zones and bad zones, with respect to a given property one wants to verify. This gives a behavioral cartography of the system. The main interest of this technique is that this behavioral cartography does not depend on the property one wants to verify: only the partition into good and bad tiles actually does. As a consequence, when verifying other properties, it is sufficient to check the property for only one point in each tile in order to get the new partition. Although the cartography may contain holes, i.e., zones not covered by the algorithm, we give sufficient condition for the full coverage of the real-valued bounded parameter domain.

In Chapter 6, we extended this method to probabilistic systems, in order to synthesize constraints on the parameters guaranteeing the preservation of probabilities for reachability properties. As for the non-probabilistic framework, it gives us a measure of *robustness* of the system. Moreover, it allows us to *rescale* the system to a valuation much smaller than the original one, thus making the probabilistic analysis of the system easier and faster in practice. This approach is also useful for avoiding repeated executions of probabilistic model checking analyses for the same model with different parameter valuations.

In Chapter 7, we also presented variants of the inverse method in two other frameworks: directed weighted graphs and Markov Decision Processes. We first introduced an algorithm allowing us to synthesize constraints on the weights of directed weighted graphs seen as parameters, guaranteeing that the shortest path in a graph remains the same, for any valuation of those parameters satisfying this constraint. We then introduced an extension of a classical algorithm for policy iteration, which synthesizes constraints on the weights of Markov decision processes seen as parameters, guaranteeing that the optimal policy in a Markov Decision Process remains the same, for any valuation of those parameters satisfying this constraint. Two prototypes, INSPEQTOR and IMPRATOR have been implemented.

As a summary of this summary, here are the main contributions of this thesis:

1. design of a new method of synthesis of timing parameters for timed automata (inverse method);

2. implementation of a tool integrating this method (IMITATOR and IMITATOR II), allowing the automated analysis of various case studies of the literature in the field of protocols and hardware circuits;

3. design and implementation of a behavioral cartography extension of the method; application to various case studies, and in particular to the memory SPSMALL designed by ST-Microelectronics;

4. extension of the method to probabilistic timed models; application, in coupling with the probabilistic model checker PRISM, to the parametric verification of various communication protocols;

5. exploration of the application of the inverse method to other domains than timed automata.

8.2 Future Research

Extension to Hybrid Systems. It would be interesting to extend the inverse method to *hybrid automata*, where clocks evolve at different rates. This formalism is of interest for the verification of electronic devices modeled at the transistor level. A first class of hybrid systems would be linear hybrid automata where, in each location, the derivatives of the variables are comprised in bounded (constant) intervals.

Backward Reasoning. The inverse method for timed automata has been designed using a *forward* analysis: starting from a state, one computes the set of states reachable in a forward manner, using the *Post* operation in Algorithm 1. It would be interesting to investigate the *backward* analysis, using the *Pre* operation. Instead of considering traces originating from an initial state, one would consider traces *reaching* a final state. The main practical application of this approach is to study the possible behavior of the system leading to a given good (or bad) state. Although no *a priori* obstacle prevents us to adapt the inverse method to the backward case, proofs (based on the forward operation) should be rewritten, and the implementation involves some tricky modifications (see paragraph below concerning the improvements to IMITATOR II).

Trace-Based Inverse Method. Instead of considering a reference valuation, one may want to consider a reference *trace* (or even *trace set*). This adaptation would consist in exploring the state space, as in the current version of our inverse method, and remove states incompatible with the reference trace, by negating inequalities. However, this appears to be more tricky than in our case, because the choice of the inequality to negate is not as straightforward. In the current version of our inverse method, it is easy to choose an incompatible inequality, because it is π_0-incompatible. In the case of a trace-based inverse method, this question needs to be investigated more deeply.

Actually, a main advantage of such an extension of the inverse method to reference traces is that it would allow to consider *partial orders*. This would be a major improvement of our method, because a weakness is that the equality of trace sets is generally a too strong requirement. One is often not interested in practice in guaranteeing the *exact* sequence of actions, but would like to allow some partial orders. An extension of the inverse method to traces allowing partial orders for a set of actions considered to be independent would be of high interest. Some recent work on Timed Automata with partial orders (see in particular [SBM06]) could be a first basis for this extension.

Preservation of Temporal Logics. So far, our inverse method in the framework of PTAs preserves the properties expressed in the linear time logics (LTL), verifiable for finite traces only (i.e., reachability and safety properties). In other words, if a reachability or safety LTL formula holds for a TA $\mathscr{A}[\pi_0]$, then it holds also for $\mathscr{A}[\pi]$, for all $\pi \models IM(\mathscr{A}, \pi_0)$. However, CTL formulae are not preserved.

A first step would be to be able to prove whether our inverse method preserves or not formulae expressed using the full LTL logics, i.e., including formulae on infinite traces. If this holds, this would allow to guarantee liveness properties. If this does not hold, it would be interesting to investigate what we should modify in the algorithm so that LTL properties on infinite traces are also preserved. Recall that we give some hints on a possible proof for the preservation of the full set of LTL properties in Remark 3.16.

It would also be interesting to investigate how modifications of the inverse method algorithm would allow to preserve CTL formulae. This is more tricky, since our inverse method is based on sets of (single) traces. On the contrary, the CTL formulae need to take the branching structure of the computation tree into account. One should as a consequence synthesize constraints preserving the branching structure, i.e., the time at which the transitions may or may not be taken.

Finally, recall that our inverse method does not preserve properties expressed in the TCTL logic for two reasons. First, TCTL is a timed extension of CTL, based on the branching structure of the system. Our inverse method does not preserve CTL, and will not preserve TCTL for the same reasons. Second, TCTL properties are not time-abstract properties, because they express facts involving time, such as "an event must occur within a 2 seconds". Nevertheless, it would be possible to guarantee some kind of timed properties, basically properties expressed LTL on finite traces and involving time, by doing as follows. One should add an observer to the model, i.e., an additional PTA modeling the timed property one is interested to verify. Then, one should apply the inverse method to the global system including the observer. The constraint output by the inverse method will then guarantee that this timed property is verified. Although interesting in theory, this approach has not been much investigated in our framework because this leads on most examples to a dramatical explosion of the state space.

Variants in the Probabilistic Framework. Recall that 3 variants of the inverse method have been designed, viz., $IM^{\subseteq}$, $IM^{\cup}$ and $IM^{\subseteq}_{\cup}$. Since these variants do not preserve the equality of trace sets anymore, the extension to probabilistic systems studied in Chapter 6 does not preserve anymore the minimum and maximum probabilities of reachability properties. It would thus be of interest to investigate if those variants still partially preserve some properties, for instance, the maximal probabilities only, or under- or over-approximations of some probabilities.

Improvement of IMITATOR II. Future work include the automatic partition into good and bad tiles, using an external tool such as UPPAAL, which can check the property one is interested in, for a given valuation of the parameters. Recall that a main advantage of the cartography method is that it does not depend on the property one wants to check. As a consequence, one should first output a cartography (i.e., a list of tiles), and then provide another command (or another tool) taking as input this cartography and the property one wants to check, and outputting the partition into good and bad tiles.

Moreover, it would be interesting to implement the variant $IM^{\cup}$ of the inverse method (the variant $IM^{\subseteq}$ has already been implemented), and consider it also when performing the behavioral cartography. The number of tiles should be much less than using the standard version *IM*.

It would also be interesting to study a "dynamic" cartography, where the space unit between the selected points (so far, one integer) can be automatically refined in order to fill the remaining holes. For example, in the Root Contention Protocol case study (see Section 6.6.2), it is possible to fill the small holes in the lower right corner by considering points multiple of 0.1 instead of integer points. Recall that it is not necessarily possible to cover completely the parameter space (even within a bounded parameter domain). As a consequence, one may want to provide a limit, so that the tool would cover as much space as possible, and at least, e.g., the rationals multiple of 0.1.

As said above, it would also be interesting to reason in a backward manner, i.e., considering a *Pre* operation instead of *Post* in Algorithm 1. A possible implementation is to reverse the description of the PTAs describing the model, and apply the standard Algorithm 1 using the *Post* operation to this reversed model. However, this restricts the use of discrete variables (allowed in the current version of IMITATOR II), because one should then only allow the use of a bijective function (on discrete variables only) when updating discrete variables.

When considering the expressiveness of the automata accepted by IMITA-

TOR II, it would be of interest to allow the use of *urgent actions* (or “as soon as possible” actions), i.e., actions that the system *must* take as soon as the guards of the different transitions synchronizing on this action are all verified. This would allow in particular to verify case studies such as the IEEE 802.11 Wireless Local Area Network Protocol (see Section 6.5.3). Various data types could also improve the expressiveness of the tool, such as arrays of integers like in tool UPPAAL.

Application to Other Formalisms. Although the inverse method has been mainly applied to the framework of timed automata, the underlying principle may also be applied to other formalisms, as shown in Chapter 7.

It would be interesting to consider an extension to *weighted* (or priced) timed automata, an extension of the class of classical timed automata allowing the use of *weights* in both locations and transitions of the automaton [ATP01, BFH$^+$01]. A classical problem in this framework is to define schedulers in order to find the optimal cost with respect to some optimization criterion. When applying the inverse method to this framework, one should introduce a parameterization of weighted timed automata, where the weights would be unknown constants or parameters (as it has been done for Directed Weighted Graphs and Markov Decision Processes in Chapter 7). Then, one should adapt the algorithms of [ATP01, BFH$^+$01] to the parametric case, in order to synthesize constraints guaranteeing that, e.g., the optimal path between any two nodes remains the same. A challenging issue would then be to combine two kinds of parameters, i.e., weight parameters (as for Parametric Directed Weighted Graphs) and timing parameters (as for Parametric Timed Automata), and adapt the inverse method to this framework.

Due to the similarity between the formalisms of Timed Automata and Time Petri Nets, it would also be interesting to investigate an application of our inverse method to (classes of) Time Petri Nets. A parametrization of Time Petri Nets has already been investigated (see, e.g., [TLR08]), and could be used as a basis for such an extension of the inverse method.

Bibliography

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *CAV '00*, pages 419–434. Springer-Verlag, 2000.

[AAC$^+$09] S. Amari, É. André, T. Chatain, O. De Smet, B. Denis, E. Encrenaz, L. Fribourg, and S. Ruel. Timed analysis of networked automation systems combining simulation and parametric model checking. Research Report LSV-09-14, Laboratoire Spécification et Vérification, ENS Cachan, France, 2009. SIMOP Research Report. 49 pages.

[ABB$^+$01] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. Uppaal - now, next, and future. In *Proc. MOVEP'00, LNCS 2067*, pages 99–124. Springer, 2001.

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *CAV '01*, pages 368–372. Springer-Verlag, 2001.

[ACD91] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for probabilistic real-time systems. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 115–126, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[ACD93] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACD$^+$09] É. André, T. Chatain, O. De Smet, L. Fribourg, and S. Ruel. Synthèse de contraintes temporisées pour une architecture d'automatisation en réseau. In Didier Lime and Olivier H. Roux, editors, *MSR'09*, volume 43 of *Journal Européen des Systèmes Automatisés*, pages 1049–1064. Hermès, 2009.

[ACEF09] É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.

[ACH$^+$92] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third International Conference on Concurrency Theory*, CONCUR '92, pages 340–354, London, UK, 1992. Springer-Verlag.

[ACHH92] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.

[AEF09] É. André, E. Encrenaz, and L. Fribourg. Synthesizing parametric constraints on various case studies using IMITATOR. Research Report LSV-09-13, Laboratoire Spécification et Vérification, ENS Cachan, France, June 2009.

[AF09] É. André and L. Fribourg. An inverse method for policy-iteration based algorithms. In Azadeh Farzan and Axel Legay, editors, *Proceedings of the 11th International Workshop on Verification of Infinite State Systems (INFINITY'09)*, volume 10 of *Electronic Proceedings in Theoretical Computer Science*, pages 44–61, Bologna, Italy, 2009.

[AF10] É. André and L. Fribourg. Behavioral cartography of timed automata. In *RP'10*, volume 6227 of *LNCS*, pages 76–90. Springer, 2010.

[AFS09] É. André, L. Fribourg, and J. Sproston. An extension of the inverse method to probabilistic timed automata. In *AVoCS'09*, volume 23 of *Electronic Communications of the EASST*, 2009.

[AHV93] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC '93*, pages 592–601. ACM, 1993.

[AKRS08] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98, New York, NY, USA, 2008. ACM.

[Alu92] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford, CA, USA, 1992.

[And09a] Étienne André. IMITATOR: A tool for synthesizing constraints on timing bounds of timed automata. In Martin Leucker and Carroll Morgan, editors, *ICTAC'09*, volume 5684 of *LNCS*, pages 336–342. Springer, 2009.

[And09b] Étienne André. Une méthode inverse pour les plus courts chemins. In *Actes de la 6ème École Temps-Réel (ETR'09)*, Paris, France, September 2009.

[And10a] Étienne André. IMITATOR II: A tool for solving the good parameters problem in timed automata. In Yu-Fang Chen and Ahmed Rezine, editors, *INFINITY'10*, volume 39 of *Electronic Proceedings in Theoretical Computer Science*, pages 91–99, 2010.

[And10b] Étienne André. IMITATOR II user manual. Research Report LSV-10-20, Laboratoire Spécification et Vérification, ENS Cachan, France, November 2010.

[And10c] Étienne André. Synthesizing parametric constraints on various case studies using IMITATOR II. Research Report LSV-10-21, Laboratoire Spécification et Vérification, ENS Cachan, France, November 2010.

[ATP01] R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. In *HSCC '01: Proceedings of the 4th International Workshop on Hybrid Systems*, pages 49–62, London, UK, 2001. Springer-Verlag.

[Bar09] Abdelrezzak Bara. Vhdl2ta: A tool for automatic translation of vhdl programs plus timings into timed automata. Research report, LIP6, 2009. ANR-VALMEM Technical Report.

[BC05] M. Baclet and R. Chevallier. Timed verification of the SPSMALL memory. In *Proceedings of the 1st International Conference on Memory Technology and Design (ICMTD'05)*, pages 89–92, Giens, France, May 2005.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.

[BDFP04] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2-3):291–345, August 2004.

[Bea03] Danièle Beauquier. On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003.

[Bel57] Richard E. Bellman. A Markov decision process. *Journal of Mathematical Mechanics*, 6:679–684, 1957.

[BFH$^+$01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. W. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, pages 147–161, 2001.

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BM83] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. In *Proceedings IFIP*, pages 41–46. Elsevier Science Publishers, 1983.

[BMR06] P. Bouyer, N. Markey, and P.-A. Reynier. Robust model-checking of linear-time properties in timed automata. In Jose R. Correa, Alejandro Hevia, and Marcos Kiwi, editors, *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, volume 3887 of *LNCS*, pages 238–249, Valdivia, Chile, March 2006. Springer.

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14), 2004.

[BS95] J. A. Brzozowski and C. J. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[BV06] B. Berthomieu and F. Vernadat. Time petri nets analysis with tina. In *QEST*, pages 123–124, 2006.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC05] R. Clarisó and J. Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *ACSD '05*. IEEE Computer Society, 2005.

[CC07] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Sci. Comput. Program.*, 64(1):115–139, 2007.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CDF$^+$08] N. Chamseddine, M. Duflot, L. Fribourg, C. Picaronny, and J. Sproston. Computing expected absorption times for parametric determinate probabilistic timed automata. In *Proc. QEST'08*, pages 254–263. IEEE, 2008.

[CDY01] S. Chakraborty, D. L. Dill, and Y. Yun. Efficient algorithms for approximate time separation of events. In *Academy Proceedings in Engineering Sciences*, 2001.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CEFX06] R. Chevallier, E. Encrenaz-Tiphène, L. Fribourg, and W. Xu. Timing analysis of an embedded memory: SPSMALL. *WSEAS Transactions on Circuits and Systems*, 5(7):973–978, July 2006.

[CEFX09] R. Chevallier, E. Encrenaz, L. Fribourg, and W. Xu. Timed verification of the generic architecture of a memory circuit using parametric timed automata. *Formal Methods in System Design*, 34(1):59–81, 2009.

[CFLY10] S. Chakraborty, E. Fischer, O. Lachish, and R. Yuster. Two-phase algorithms for the parametric shortest path problem. In Jean-Yves Marion and Thomas Schwentick, editors, *Proceedings of the 27th Annual Symposium on the Theoretical Aspects of Computer Science*

(STACS 2010), pages 167–178, Nancy France, 2010. Inria Nancy Grand Est & Loria.

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

[CHR02] F. Cassez, T. A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *HSCC '02: Proceedings of the 5th International Workshop on Hybrid Systems: Computation and Control*, pages 134–148, London, UK, 2002. Springer-Verlag.

[CPR08] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 80–89, Washington, DC, USA, 2008. IEEE Computer Society.

[CR06] F. Cassez and O. H. Roux. Structural translation from Time Petri Nets to Timed Automata – Model-Checking Time Petri Nets via Timed Automata. *The journal of Systems and Software*, 79(10):1456–1468, 2006.

[CS01] A. Collomb–Annichini and M. Sighireanu. Parameterized reachability analysis of the IEEE 1394 Root Contention Protocol using TReX. In *RT-TOOLS '01*, 2001.

[CSD97] S. Chakraborty, P. A. Subrahmanyam, and D. L. Dill. Approximate time separation of events in practice. In *Proc. of 5th ACM/IEEE Int. Workshop TAU (REFERENCE A REVOIR)*, 1997.

[CTCG$^+$98] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. Mc Gettrick, and J-P. Quadrat. Numerical computation of spectral elements in max-plus algebra. In *IFAC Conf. on Syst. Structure and Control*, 1998.

[Daw04] Conrado Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Proc. ICTAC'04*, volume 3407 of *LNCS*, pages 280–294. Springer, 2004.

[DDMR04] M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robustness and implementability of timed automata. In Yassine Lakhnech and Sergio Yovine, editors, *Proceedings of the Joint Conferences Formal Modelling and Analysis of Timed Systems (FORMATS'04) and Formal Techniques in Real-Time and Fault-Tolerant Systems*

(FTRTFT'04), volume 3253 of *LNCS*, pages 118–133, Grenoble, France, 2004. Springer.

[DDSS07] D. D'Aprile, S. Donatelli, A. Sangnier, and J. Sproston. From time Petri nets to timed automata: An untimed approach. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 216–230, Braga, Portugal, March 2007. Springer.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[DKRT97] P.R. D'Argenio, J.P. Katoen, T.C. Ruys, and G.J. Tretmans. The bounded retransmission protocol must be on time! In *TACAS '97*. Springer, 1997.

[Doy07] Laurent Doyen. Robust parametric reachability for timed automata. *Information Processing Letters*, 102(5):208–213, 2007.

[DRF$^+$07] B. Denis, S. Ruel, J.-M. Faure, G. Marsal, and G. Frey. Measuring the impact of vertical integration on response times in Ethernet fieldbuses. In *Proc. of ETFA'07*, 2007.

[DWDR05] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost asap semantics: from timed models to timed implementations. *Form. Asp. Comput.*, 17(3):319–341, 2005.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *IEEE Real-Time Systems Symposium*, pages 66–75, 1995.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[EF08] E. Encrenaz and L. Fribourg. Time separation of events : An inverse method. In *Proceedings of the LIX Colloquium '06*, volume 209 of *ENTCS*, Palaiseau, France, 2008. Elsevier Science Publishers.

[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.

[FKR06] G. Frehse, B.H. Krogh, and R.A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 257–262. European Design and Automation Association, 2006.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6), 1962.

[Fre05a] Goran Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Ph.d. thesis, Radboud University Nijmegen, 2005.

[Fre05b] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *HSCC*, pages 258–273, 2005.

[GHJ97] V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In Oded Maler, editor, *Proceedings of the 1997 International Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *LNCS*, pages 331–345. Springer-Verlag, March 1997.

[GJ95] H. Gregersen and H. E. Jensen. Formal design of reliable real time systems. Master's thesis, Department of Mathematics and Computer Science, Aalborg University, 1995.

[GPSS80] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In *POPL*, pages 163–173, 1980.

[gWp] Graphviz Web page. `http://www.graphviz.org/`.

[HH07] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[HHWT95] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. A user guide to HyTech. In *TACAS*, pages 41–71, 1995.

[HHWT97] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[HHWZ10] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PARAM: A model checker for parametric markov models. In *CAV*, pages 660–664, 2010.

[HKM08] T. Han, J.-P. Katoen, and A. Mereacre. Approximate parameter synthesis for probabilistic time-bounded reachability. In *Proc. RTSS'08*, pages 173–182. IEEE, 2008.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS'06*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[Hol03] Gerard Holzmann. *The Spin model checker: primer and reference manual.* Addison-Wesley Professional, 2003.

[How60] R. A. Howard. *Dynamic Programming and Markov Processes.* John Wiley and Sons, Inc., 1960.

[HRSV02] T.S. Hune, J.M.T. Romijn, M.I.A. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 2002.

[HSLL97] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[HWT96] T. A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS 1165.* Springer-Verlag, 1996.

[JM09] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV '09*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[KMST59] J. Kemeny, H. Mirkil, J. Snell, and G. Thompson. *Finite mathematical structures.* Prentice-Hall, Englewood Cliffs, N.J., 1959.

[KNP09] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic games for verification of probabilistic timed automata. In J. Ouaknine and F. Vaandrager, editors, *FORMATS'09*, volume 5813 of *LNCS*, pages 212–227. Springer, 2009.

[KNPS06] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Form. Methods Syst. Des.*, 29:33–78, 2006.

[KNS02] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In *Proc. PAPM/PROBMIV'02*, volume 2399 of *LNCS*, pages 169–187. Springer, 2002.

[KNS03] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

[KNSS02] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282:101–150, 2002.

[KNSW07] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, 2007.

[KP10] M. Knapik and W. Penczek. Bounded model checking for parametric time automata. In *SUMo '10*, 2010.

[KSK76] J. G. Kemeny, J. L. Snell, and A. W Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 2nd edition, 1976.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[LMST07] R. Lanotte, A. Maggiolo-Schettini, and A. Troina. Parametric probabilistic transition systems for system design and analysis. *Form. Asp. Comput.*, 19(1):93–109, 2007.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Logic of Programs*, pages 196–218, 1985.

[LRST09] D. Lime, O. H. Roux, C. Seidner, and L.-M. Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In Stefan Kowalewski and Anna Philippou, editors, *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *LNCS*, pages 54–57, York, United Kingdom, March 2009. Springer.

[LY93] K. G. Larsen and W. Yi. Time abstracted bisimiulation: Implicit specifications and decidability. In *MFPS*, pages 160–176, 1993.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mer74] Philip Meir Merlin. *A study of the recoverability of computing systems*. PhD thesis, 1974.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *CHARME '95*, pages 189–205. Springer-Verlag, 1995.

[MY96] O. Maler and S. Yovine. Hardware timing verification using kronos. In *ICCSSE '96: Proceedings of the 7th Israeli Conference on Computer-Based Systems and Software Engineering*, page 23, Washington, DC, USA, 1996. IEEE Computer Society.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[PBBDM98] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli. Policy optimization for dynamic power management. In *DAC '98*, pages 182–187, New York, NY, USA, 1998. ACM.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, 1962.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PP06] W. Penczek and A. Pólrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach (Studies in Computational Intelligence).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Pur00] Anuj Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[pWpa] gnuplot Web page. `http://www.gnuplot.info/`.

[pWpb] PRISM Web page. Prism web page.

[RDS08] S. Ruel and J.-M. De Smet, O. Faure. Efficient representation for formal verification of time performances of networked automation architectures. In *Proc. of 17th IFAC World Congress*, pages 5119–5124, July 2008.

[ROC05] R. Ramaswamy, J. B. Orlin, and N. Chakravarti. Sensitivity analysis for shortest path problems and maximum capacity path problems in undirected graphs. *Math. Program., Ser. A*, 102:355–369, 2005.

[SB03] L. Stachniss and W. Burgard. The markov decision problem - autonomous mobile systems. Course notes, University Freiburg, Germany, 2003.

[SBM06] R. Ben Salah, M. Bozga, and O. Maler. On interleaving in timed automata. In *CONCUR '06*, volume 4137 of *LNCS*, pages 465–476. Springer, 2006.

[Sch86] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, 1995.

[Sou10a] Romain Soulat. Améliorations algorithmiques d'un moteur de model-checking et études de cas. Rapport de Master, Master 2 Recherche Informatique Paris Sud 11, 2010.

[Sou10b] Romain Soulat. Analysis of the bounded retransmission protocol using IMITATOR II. 2010.

[Sou10c] Romain Soulat. On properties of the inverse method: Commutation of instantiation. Research Report LSV-10-22, Laboratoire Spécification et Vérification, ENS Cachan, France, November 2010.

[Spr01] Jeremy Sproston. *Model Checking for Probabilistic Timed and Hybrid Systems.* PhD thesis, School of Computer Science, University of Birmingham, 2001.

[SRD09] P. Bazargan Sabet, P. Renault, and D. Le Dû. Prototype d'outil d'abstraction fonctionnelle, 2009. VALMEM Project deliverable 2.4.

[TLR08] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of time petri nets with stopwatches using the state-class graph. In *FORMATS '08: Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, pages 280–294, Berlin, Heidelberg, 2008. Springer-Verlag.

[TY98] S. Tripakis and S. Yovine. Verification of the fast reservation protocol with delayed transmission using the tool kronos. In *IEEE Real Time Technology and Applications Symposium*, pages 165–, 1998.

[TY01] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[vWp] VHDL2TA Web page. `http://www.lsv.ens-cachan.fr/~encrenaz/valmem/vhdl2hytech/`.

[Wan06] Farn Wang. REDLIB for the formal verification of embedded systems. In *ISoLA*, pages 341–346, 2006.

[WEE$^+$08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[WY03] F. Wang and H.C. Yen. Timing parameter characterization of real-time systems. In *CIAA '03*, volume 2759 of *LNCS*, pages 23–34, 2003.

[YKM02] T. Yoneda, T. Kitai, and C. J. Myers. Automatic derivation of timing constraints by failure analysis. In *CAV '02*, pages 195–208. Springer-Verlag, 2002.

[YL97] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. *Form. Methods Syst. Des.*, 11:113–136, 1997.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

[ZC05] D. Zhang and R. Cleaveland. Fast on-the-fly parametric real-time model checking. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 157–166, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Classical Notions

A.1 LTL

We recall below the syntax and semantics of the Linear Temporal Logic (LTL) [Pnu77]. We borrow most of the following from [BK08].

Syntax. Given a set *AP* of atomic propositions, LTL formulae over the set *AP* are formed according to the following grammar:

$$\varphi ::= \texttt{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2$$

where $a \in AP$.

The $\bigcirc$ modality, pronounced "next", is a unary prefix operator requiring a single LTL formula as argument. Formula $\bigcirc\varphi$ holds at the current moment if φ holds in the next step. The $\cup$ modality, pronounced "until", is a binary infix operator requiring two LTL formula as argument. Formula $\varphi_1 \cup \varphi_2$ holds at the current moment if there is some future moment for which φ_2 holds and φ_1 holds at all moment until this future moment.

Using the Boolean connectives $\wedge$ and $\neg$, the full power of propositional logic is obtained. One can derive the other booleans connectives, such as disjunction $\vee$, implication $\rightarrow$, or equivalence $\leftrightarrow$, in a classical way.

The until operator allows to derive the temporal modalities $\Diamond$ ("eventually", sometimes in the future) and $\Box$ ("always", from now on forever) as follows:

$$\begin{aligned} \Diamond &\overset{def}{=} \texttt{true} \cup \varphi \\ \Box &\overset{def}{=} \neg\Diamond\neg\varphi \end{aligned}$$

As a result, $\Diamond\varphi$ ensures that φ will be true eventually in the future. Moreover, $\Box\varphi$ is satisfied if and only if it is not the case that eventually $\neg\varphi$ holds. This is equivalent to the fact that φ holds from now on forever.

By combining the temporal modalities $\Diamond$ and $\Box$, new temporal modalities are obtained, such as $\Box\Diamond a$ ("always eventually a") or $\Diamond\Box a$ ("eventually forever a").

Semantics. The semantics of LTL formula φ is defined as a language *Words*(φ) that contains all infinite words over the alphabet 2^{AP} that satisfy φ. Let φ be an LTL formula over *AP*. The linear time property induced by φ is

$$Words(\varphi) = \left\{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\right\}$$

where the satisfaction relation $\models \subseteq (2^{AP})^\omega \times LTL$ is the smallest relations with the properties in Figure A.1.

σ	$\models$	`true`		
σ	$\models$	a	iff	$a \in A_0$ (i.e., $A_0 \models a$)
σ	$\models$	$\varphi_1 \wedge \varphi_2$	iff	$\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
σ	$\models$	$\neg\varphi$	iff	$\sigma \not\models \varphi$
σ	$\models$	$\bigcirc\varphi$	iff	$\sigma[1\ldots] = A_1A_2A_3\ldots \models \varphi$
σ	$\models$	$\varphi_1 \cup \varphi_2$	iff	$\exists j \geq 0.\sigma[j\ldots] \models \varphi_2$ and $\sigma[i\ldots] \models \varphi_1$, for all $0 \leq i \leq j$

Figure A.1: LTL semantics for infinite words over 2^{AP}

Here, for $\sigma = A_0A_1A_2\cdots \in (2^{AP})^\omega$, $\sigma[j\ldots] = A_jA_{j+1}A_{j+2}\ldots$ is the suffix of σ starting in the $(j+1)$st symbol A_j.

For the derived operators $\Diamond$ and $\Box$, we have:

$$\begin{array}{llll} \sigma & \models & \Diamond\varphi & \text{iff} \quad \exists j \geq 0.\sigma[j\ldots] \models \varphi \\ \sigma & \models & \Box\varphi & \text{iff} \quad \forall j \geq 0.\sigma[j\ldots] \models \varphi. \end{array}$$

We do not give here the semantics or the combinations of $\Diamond$ and $\Box$, which can be defined in a straightforward way and found, e.g., in [BK08].

A.2 CTL

We recall here the syntax and semantics of CTL [CE82] (Computation Tree Logic), which is a widely used branching temporal logic. We borrow most of the following from [BK08].

Syntax. CTL has a two-stage syntax where formulae are classified into state ant path formulae. The notion of "path" is classical in the literature, and we will stick to it. Note nevertheless that those paths correspond to the *concrete runs* of our PTAs.

In our framework, a major difference with LTL is that it is not sufficient to consider traces (or trace sets) of PTAs; one has to consider the *timed* semantics of PTAs, i.e., their full semantics as a LTS.

CTL state formulae over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi ::= \texttt{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula.

CTL path formulae are formed according to the following grammar:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \cup \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae.

The temporal operators $\bigcirc$ and $\cup$ have the same meaning as in LTL and are path operators. Path formulae can be turned into state formulae by prefixing them with either the path quantifier $\exists$ (pronounced "for some path") or the path quantifier $\forall$ (pronounced "for all path"). Formula $\exists\varphi$ holds in a state if there exists some path satisfying φ that starts in that state. Dually, formula $\forall\varphi$ holds in a state if all paths that start in that state satisfy φ.

Semantics. CTL formulae are interpreted over the states and paths of an LTS. Given an LTS, the semantics of CTL formulae is defined by two satisfaction relations (both denoted by $\models$): one for the state formulae, and one for the path formulae. For the state formulae, $\models$ is a relation between the states of the LTS and state formulae. We have that $s \models \Phi$ if and only if state formula Φ holds in state s. For the path formulae, $\models$ is a relation between maximal run fragments of the LTS and path formulae. We have that $R \models \varphi$ if and only if run R satisfies path formula φ.

Let $a \in AP$ be an atomic proposition, $\mathscr{L} = (S, S_0, \Rightarrow)$ be an LTS corresponding to the concrete semantics of a PTA, state $s \in S$, Φ, Φ_1 and Φ_2 be CTL state formulae, and ϕ be a CTL path formula. Recall that we consider that the set of the atomic propositions correspond to the locations of the PTAs. As a consequence, we assume that the labeling function L assigns to a concrete state the corresponding location, i.e.? is such that $L(q, w) = q$. We denote by $R \in \mathit{Runs}(s)$ the set of runs starting from state s in the LTS. The satisfaction relation $\models$ is defined for state formulae by:

$$
\begin{array}{llll}
s & \models & a & \text{iff} \quad a = L(s) \\
s & \models & \neg\Phi & \text{iff} \quad \text{not } s \models \Phi \\
s & \models & \Phi_1 \wedge \Phi_2 & \text{iff} \quad s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s & \models & \exists\varphi & \text{iff} \quad R \models \varphi \text{ for some } R \in \mathit{Runs}(s) \\
s & \models & \forall\varphi & \text{iff} \quad R \models \varphi \text{ for all } R \in \mathit{Runs}(s)
\end{array}
$$

For a run R, the satisfaction relation $\models$ for path formulae is defined by

$$
\begin{array}{llll}
R & \models & \bigcirc\Phi & \text{iff} \quad R[1] \models \Phi \\
R & \models & \Phi_1 \cup \Phi_2 & \text{iff} \quad \exists j \geq 0 \text{ s.t. } R[j] \models \Phi_2 \wedge (\forall 0 \leq k \leq j : R[k] \models \Phi_1)
\end{array}
$$

where for run $R = s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} \cdots$ and integer $i \geq 0$, $R[i]$ denotes the $(i+1)$th state of R, i.e., $R[i] = s_i$.

A.3 Time-Abstract Bisimulation

We recall here the notion of time-abstract bisimulation, a relation which is insensitive to time-quantities. This notion (defined, e.g., in [LY93]) is defined below in our framework of semantics of TAs.

Let $\mathscr{A}_1$ and $\mathscr{A}_2$ be two TAs. Let $\mathscr{L}_1$ (resp. $\mathscr{L}_2$) be the LTS corresponding to the concrete semantics of $\mathscr{A}_1$ (resp. $\mathscr{A}_2$). A binary relation $\mathscr{R}$ between $\mathscr{L}_1$ and $\mathscr{L}_2$ is a strong timed simulation if $(s_1, s_2) \in \mathscr{R}$ implies that for all $a \in \Sigma$, we have that whenever $s_1 \overset{a}{\Rightarrow} s_1'$ then, for some s_2', $s_2 \overset{a}{\Rightarrow} s_2'$ and $(s_1', s_2') \in \mathscr{R}$.

We call such a simulation $\mathscr{R}$ a strong timed bisimulation if it is symmetrical.

Notations

We recall below the most frequent letters used throughout (or in part of) this thesis.

Latin Letters

C	Constraint on the clock and the parameters
d	constant $\in \mathbb{R}_{\geq 0}$
D	constraint on the clocks
e	linear term
g	guard
H	cardinality of the set of clocks
I	invariant
J	inequality on the parameters
K	constraint on parameters
M	cardinality of the set of parameters
p	parameter
P	set of parameters
q	location
Q	set of locations
r	run
s	state
S	set of states
w	clock valuation / weight function (MDP)
W	parametric weight function (MDP)
x	clock
X	set of clocks

Cursive Letters

$\mathscr{A}$	TA / PTA / probabilistic TA / PPTA
$\mathscr{C}$	matrix of costs (Section 7.1)

$\mathscr{D}$	parametric matrix of costs (Section 7.1)
$\mathscr{G}$	DWG / PDWG (Section 7.1)
$\mathscr{K}_X$	constraint on the clocks
$\mathscr{K}_P$	constraint on the parameters
$\mathscr{K}_{X \cup P}$	constraint on the clocks and the parameters
$\mathscr{L}$	LTS
$\mathscr{M}$	MDP (Section 7.2)
$\mathscr{S}$	successor matrix (Section 7.1)
$\mathscr{V}$	shortest path matrix (Section 7.1)
$\mathscr{W}$	parametric shortest path matrix (Section 7.1)

Greek Letters

α	coefficient for variable
μ	distribution (Chapter 6)
ν	policy (Section 7.2)
π	valuation of the parameters
ρ	set of clock variables to be reset
σ	scheduler (Chapter 6)
Σ	set of actions
φ	LTL / CTL formula
ω	path of a TPS (Chapter 6)
Ω	set of paths (Chapter 6)

Sets of Numbers

$\mathbb{N}$	set of non-negative integers
$\mathbb{Q}$	set of rational numbers
$\mathbb{Q}_{\geq 0}$	set of non-negative rational numbers
$\mathbb{R}$	set of real numbers
$\mathbb{R}_{\geq 0}$	set of non-negative real numbers

Glossary

CTL
Computation Tree Logic (Appendix A.2)

DWG
Directed Weighted Graph (Definition 7.1)

LTL
Linear Time Logic (Appendix A.1)

LTS
Labeled Transition System (Definition 2.7)

MDP
Markov Decision Process (Definition 7.16)

NAS
Networked Automation System (Section 4.8)

NPTA
Network of Parametric Timed Automata (Definition 2.25)

NTA
Network of Timed Automata (Definition 2.10)

PDWG
Parametric Directed Weighted Graph (Definition 7.8)

PMDP
Parametric Markov Decision Process (Definition 7.25)

PPTA
Parametric Probabilistic Timed Automaton (Definition 6.13)

PTA
Probabilistic Timed Automaton (Definition 2.21)

TCTL
Timed Computation Tree Logic (Definition 2.7)

TPS
Timed Probabilistic System (Definition 6.1)

Index

Résumé substantiel

Introduction

Contexte. L'importance des systèmes informatiques dans la société a crû de façon drastique depuis plusieurs décennies. Les systèmes critiques, impliquant des vies humaines, doivent être parfaitement stables, et ne pas mener au moindre comportement inapproprié. Ces comportements inappropriés correspondent par exemple à des erreurs ou des séquences d'actions imprévues. Il est possible de tester la correction d'un système afin de vérifier l'absence de comportement inapproprié pour un environnement précis, en exécutant directement le système. Néanmoins, bien que le test d'un système soit à même de garantir l'absence de comportement inapproprié pour une exécution particulière, aucune garantie n'est donnée plus généralement, pour d'autres scenarii de l'environnement. De plus, pour peu qu'il y ait du non-déterminisme dans l'exécution, c'est-à-dire une part de choix indépendant de l'utilisateur, la correction d'un cas de test ne donne pas même de garantie pour d'autres exécutions correspondant au même environnement que celui de ce cas de test. C'est pourquoi des techniques formelles de vérification sont nécessaires, permettant de prouver la correction d'un système vis-à-vis d'une propriété donnée, et ce à l'aide de modèles et de preuves mathématiques.

Lorsque l'on vérifie formellement un système temps-réel, comme un circuit ou un protocole de communication, il est important de vérifier non seulement l'aspect fonctionnel, mais également *temporisé*. La correction du système dépend des valeurs de constantes temporelles internes et de l'environnement.

Les méthodes formelles de vérification garantissent la correction d'un système temporisé pour un ensemble de constantes temporelles, mais ne donnent pas d'information pour d'autres constantes. Vérifier la correction d'un système pour de nombreuses constantes peut s'avérer long et difficile. Il est alors intéressant de raisonner paramétriquement, en considérant que ces constantes sont inconnues, c'est-à-dire des *paramètres*. Le problème des bons paramètres consiste alors à synthétiser de bonnes valeurs de ces paramètres, c'est-à-dire des valeurs pour lesquelles le système a un *bon comportement*. Nous nous intéressons ici à la synthèse de paramètres dans le cadre des automates temporisés, un modèle utilisé pour la vérification de systèmes temps-réel.

Contributions. Cette thèse propose une nouvelle approche pour la synthèse de constantes temporelles dans les systèmes temporisés. Notre approche est basée sur la *méthode inverse* suivante : nous considérons que les constantes temporelles du système sont des paramètres. Partant d'une instance de référence des paramètres, nous synthétisons une contrainte sur les paramètres, garantissant le même comportement que pour l'instance de référence, abstraction faite du temps. Il en résulte un critère de robustesse pour le système, dans le sens où le système se comportera de la même manière pour des constantes autour de l'instance de référence, tant que l'on reste dans la contrainte synthétisée. En itérant cette méthode sur un ensemble de points dans un domaine paramétrique borné, nous sommes alors à même de partitionner l'espace des paramètres en bonnes et mauvaises zones par rapport à une propriété que l'on souhaite vérifier. Ceci nous donne une *cartographie comportementale* du système.

Cette méthode s'étend aisément aux systèmes probabilistes. Nous présentons également des variantes de la méthode inverse dans deux autres cadres : les graphes orientés valués, et les processus de décision markoviens. Plusieurs prototypes ont été implémentés ; en particulier, IMITATOR II implémente la méthode inverse et la cartographie dans le cadre des automates temporisés. Ce prototype nous a permis de synthétiser des valeurs de bon fonctionnement pour les paramètres temporels de plusieurs études de cas, notamment un modèle abstrait d'une mémoire commercialisée par le fabricant de puces ST-Microelectronics, ainsi que plusieurs protocoles de communication.

Préliminaires

Contraintes. On suppose donnés un ensemble de variables d'*horloges*, et un ensemble de constantes inconnues, ou *paramètres*.

On suppose ici qu'une *contrainte* est une conjonction d'inégalités linéaires sur les horloges, ou sur les paramètres, ou sur les horloges et les paramètres. Les coefficients des variables sont supposés rationnels. On autorise, pour les contraintes sur les horloges uniquement, l'usage de constantes dans les inégalités (hors coefficients).

Automates temporisés. Nous considérons ici le modèle des *automates temporisés* [AD94]. Les automates temporisés sont une extension au cas temporisé des automates d'états finis, autorisant l'usage d'*horloges*, c'est-à-dire de variables évoluant linéairement avec le temps. Classiquement, ces automates sont constitués d'un ensemble d'états de contrôle, d'un alphabet d'actions, et d'un ensemble de transitions. De plus, certaines des horloges peuvent être remises à zéro lors des transitions entre deux états de contrôle de l'automate. On associe également aux états de contrôle un *invariant*, c'est-à-dire une contrainte sur les horloges qui doit être vérifiée pour rester dans l'état de contrôle en question. Enfin, on associe également aux transitions entre deux états de contrôles une *garde*, c'est-à-dire une contrainte sur les horloges qui doit être vérifiée pour prendre la transition. Les invariants et les gardes comparent donc des horloges à des constantes, dites constantes temporelles. Ces constantes temporelles ont une grande importance, et décident généralement du comportement du système.

La sémantique des automates temporisés s'entend en termes d'*états concrets*, c'est-à-dire de couples constitués d'un état de contrôle et d'une valeur pour chaque horloge. Le comportement d'un automate est alors représentable par une *exécution concrète*, ou séquence possiblement infinie d'états concrets et de transitions discrètes entre ces états concrets. Bien entendu, la valeur de chaque horloge dans chaque état concret doit satisfaire l'invariant de l'état de contrôle associé, et deux états concrets adjacents dans l'exécution concrète doivent satisfaire la garde et les horloges remises à zéro associées à la transition entre ces deux états concrets.

On s'intéresse ici à la notion de *trace*, qui est en fait une sémantique abstraction faite du temps : une trace correspond à une exécution concrète dont on abstrait la valeur des horloges, autrement dit une séquence possiblement infinie d'états de contrôles et de transitions discrètes. L'automate temporisé est alors caractérisé par son ensemble de traces, ou l'ensemble des traces associées aux exécutions partant de l'état de contrôle initial de l'automate.

Parmi les avantages des automates temporisés, on retiendra que de nombreux problèmes sont *décidables*, notamment l'accessibilité d'un état concret. De plus, plusieurs logiques peuvent exprimer des propriétés sur les automates temporisés. Les logiques temporelles LTL [Pnu77, GPSS80, LPZ85, Lam94], à temps linéaire, ou CTL [CE82] , à temps branchant, permettent de spécifier et vérifier des propriétés associées aux traces. En outre, la logique temporelle *temporisée* TCTL [ACD93] permet de spécifier des propriétés temporisées associées aux exécutions de l'automate.

Plusieurs outils permettent de vérifier des systèmes modélisés à l'aide d'automates temporisés ou leurs extensions, notamment HyTech [HHWT97], Uppaal [LPY97], Kronos [Yov97], TReX [ABS01] ou encore PHAVer [Fre05b].

Automates temporisés paramétrés. Ajuster la valeur des constantes temporelles d'un automate temporisé peut s'avérer long et difficile : il est donc souvent intéressant de considérer que ces constantes sont des valeurs inconnues, ou *paramètres*. Nous rappelons ici l'extension paramétrée des automates temporisés. Ces *automates temporisés paramétrés* [AHV93] autorisent dans les gardes et invariants de l'automate l'utilisation de *paramètres* au lieu des constantes rationnelles.

Pour une *instance* donnée des paramètres, c'est-à-dire l'assignation à chaque paramètre d'une constante, on retrouve alors un automate temporisé classique.

La sémantique s'entend alors en termes d'*états symboliques*, c'est-à-dire de couples constitués d'un état de contrôle et d'une contrainte sur les horloges et les paramètres. Le comportement d'un automate est alors représentable par une *exécution symbolique*, ou séquence possiblement infinie d'états concrets et de transitions discrètes entre ces états symboliques. Bien entendu, la contrainte associée à chaque état symbolique doit satisfaire l'invariant de l'état de contrôle associé, et deux états symboliques adjacents dans l'exécution symbolique doivent satisfaire la garde et les horloges remises à zéro associées à la transition entre ces deux états symboliques.

Comme pour les automates temporisés classiques, nous définissons une trace comme une exécution symbolique dont on abstrait la valeur des horloges et des paramètres, autrement dit une séquence possiblement infinie d'états de contrôles et de transitions discrètes.

Une méthode inverse pour les automates temporisés

Nous nous intéressons ici au problème inverse suivant. « Soit un automate temporisé paramétré et une instance de référence des paramètres temporels. Quelles sont les autres valeurs des paramètres temporels pour lesquelles le comportement non-temporisé de l'automate est le même, c'est-à-dire telles que les ensembles de traces sont égaux ? » Ce problème permettra plus tard de nous aider à résoudre le problème des bons paramètres pour les automates temporisés.

Méthode inverse. La *méthode inverse* que nous définissons [ACEF09] pour résoudre le problème inverse peut se résumer comme suit. La méthode prend en entrée un automate temporisé paramétré $\mathscr{A}$ et une instance de référence π_0 de tous les paramètres.

Partant d'une contrainte K égale à vrai, nous calculons de manière itérative un ensemble d'états symboliques accessibles. Quand un état dit π_0-

incompatible est généré, c'est-à-dire quand la projection sur les paramètres de la contrainte sur les horloges et paramètres associée à l'état n'est pas vérifiée par π_0, on raffine alors K comme suit : on sélectionne dans K une inégalité elle-même π_0-incompatible, on la nie, et on l'ajoute à K. Cette procédure est alors réitérée avec ce nouveau K, et ainsi de suite jusqu'à ce qu'aucun nouvel état ne soit généré. Finalement, on retourne l'intersection K_0 de la projection sur les paramètres des contraintes associées à tous les états accessibles.

Les deux étapes majeures de cet algorithme sont les suivantes :

1. la négation des états π_0-incompatibles (en niant des inégalités π_0-incompatibles) garantit l'absence de traces non présentes sous π_0 ;
2. l'intersection finale de toutes les contraintes sur les paramètres associées aux états accessibles garantit que *toutes* les traces sous π_0 seront également présentes pour toute autre instance des paramètres prise dans K_0.

Correction et terminaison. Nous montrons tout d'abord [ACEF09] que, pour une contrainte K_0 retournée par notre méthode pour un automate $\mathscr{A}$ et une instance π_0, les ensembles de traces de $\mathscr{A}$ sous π_0 et $\mathscr{A}$ sous π sont égaux, quel que soit π pris dans K_0. En d'autres termes, pour toute trace finie de $\mathscr{A}$ sous π_0, il existe une trace égale dans $\mathscr{A}$ sous π et réciproquement.

La méthode ne termine pas dans le cas général. Néanmoins, nous donnons des critères de terminaison, notamment pour des automates acycliques, c'est-à-dire ne passant jamais deux fois par le même état de contrôle. En outre, en pratique, la méthode inverse termine pour la très grande majorité de nos exemples.

Propriétés. En raison de la sélection aléatoire des contraintes π_0-incompatibles et de l'inégalité π_0-incompatible au sein d'une contrainte, la méthode n'est pas *confluente* : pour une même entrée, la contrainte retournée par la méthode inverse n'est pas nécessairement toujours la même. Cette non-confluence implique une *non-maximalité* : la contrainte retournée par la méthode n'est pas nécessairement la plus large résolvant le problème inverse. En d'autres termes, il peut exister d'autres instances de paramètres en dehors de K_0 correspondant au même ensemble de traces que sous π_0.

Avantages. La méthode inverse présente les avantages suivants. Tout d'abord, elle permet de donner un critère de *robustesse* au système, en garantissant la correction du système non seulement pour une instance des paramètres, mais pour d'autres instances *autour* de cette instance nominale. Ceci se révèle intéressant lors de l'implémentation réelle d'un modèle, puisque les constantes temporelles, souvent entières, ne sont pas toujours exactement égales à des entiers, mais peuvent légèrement varier. En outre, notre méthode permet au concepteur d'un système temps-réel d'*optimiser* certaines des constantes temporelles, sans changer le comportement global du système. Enfin, la méthode inverse est une méthode *exacte*, qui n'utilise donc aucune approximation.

Variantes. Nous définissons une première variante de la méthode inverse, qui consiste à modifier le point fixe. Au lieu d'arrêter l'algorithme lorsque tous les états générés ont déjà été rencontrés par le passé, on assouplit cette condition, et on définit le point fixe comme suit : l'algorithme s'arrête lorsque tous les états générés sont *inclus* dans d'autres états déjà été rencontrés par le passé, c'est-à-dire que la projection de la contrainte sur les paramètres associée à ceux-là est incluse dans la projection de la contrainte sur les paramètres associée à ceux-ci. Cette variante permet une terminaison plus fréquente, mais également plus rapide. En revanche, la correction de la méthode n'est plus préservée : les ensembles de trace ne sont plus nécessairement égaux. Néanmoins, nous montrons que la contrainte est plus faible (donc plus large) que pour la méthode inverse standard, et la non-accessibilité d'un état de contrôle est préservée. En d'autres termes, si un état de contrôle n'appartient à aucune trace sous l'instance de référence, alors il n'appartiendra non plus à aucune trace de toute instance prise dans la contrainte générée par cette première variante. Cette propriété est tout à fait intéressante lorsque l'on s'intéresse à la sûreté d'un système.

Nous définissons une seconde variante de la méthode inverse, qui consiste à modifier la contrainte retournée. Au lieu de retourner l'intersection de la projection sur les paramètres des contraintes associées à tous les états accessibles, on retourne l'*union* de la projection sur les paramètres des contraintes associées à certains états accessibles, en l'occurrence le dernier état de chaque exécution symbolique. Puisque le point fixe n'est pas modifié, la terminaison de la méthode est la même que pour la méthode inverse standard. En revanche, nous montrons que la contrainte est plus faible (donc plus large) que pour la méthode inverse standard. En outre, bien que l'égalité des ensembles de traces ne soit pas non plus préservée ici, nous garantissons l'*inclusion* des ensembles de traces : toute trace sous une instance prise dans la contrainte générée par cette seconde variante est également une trace sous l'instance de référence (mais la réciproque n'est pas vraie). Par conséquent, la non-accessibilité d'un état de contrôle est préservée.

Une dernière variante consiste à combiner ces deux variantes, ce qui permet une meilleure terminaison, une contrainte plus large, et la préservation de la non-accessibilité d'un état de contrôle.

Aucune de ces variantes n'est pour autant ni confluente ni maximale.

Cartographie comportementale

Le problème des bons paramètres. La méthode inverse permet de généraliser une instance de référence des paramètres en garantissant le même comportement, abstraction faite du temps. Or, lorsque l'on cherche à synthétiser des valeurs de constantes temporelles correspondant à un bon comportement, on s'intéresse en général non à *un* bon comportement donné, mais à *plusieurs* voire à tous les bons comportements. Par conséquent, nous allons chercher à utiliser la méthode inverse afin de synthétiser des valeurs de constantes temporelles correspondant à *tous* les bons comportements d'un système. Le problème auquel on s'intéresse ici est le suivant. « Soit un automate temporisé et un domaine paramétrique borné. Quelles sont les valeurs des paramètres pour lesquelles le système suit un bon comportement ? » Cette notion de bon comportement s'entend relativement à une propriété sur les ensembles de traces. Par conséquent, cette propriété doit être linéaire et non temporisée, puisque les traces sont des exécutions dont on a abstrait les informations temporelles. C'est notamment le cas des propriétés exprimées avec la logique LTL sur les traces finies.

La méthode de cartographie comportementale. En itérant notre méthode inverse sur les points entiers d'un domaine paramétrique borné, nous sommes à même de partitionner l'espace paramétrique en *tuiles comportementales*, c'est-à-dire en zones paramétriques denses pour lesquelles le comportement est uniforme. En d'autres termes, les ensembles de traces sont les mêmes pour toutes les instances de paramètres dans chacune des tuiles, par propriété de la méthode inverse. Ceci nous donne donc une *cartographie comportementale* du système [AF10].

En pratique, ce qui est généralement couvert par l'algorithme n'est pas seulement le sous-espace entier et borné au sein du domaine paramétrique de départ, mais deux extensions intéressantes. D'une part, les tuiles sont *denses* et couvrent donc une large zone *réelle* du domaine de départ. D'autre part, les tuiles sont souvent non-bornées, et couvrent généralement une grande partie de l'espace paramétrique *au-delà* du domaine borné de départ.

Une fois cette couverture de l'espace paramétrique effectuée, étant donnée une propriété sur les traces que l'on souhaite vérifier, il est facile de partitionner l'espace paramétrique en « bonnes » et « mauvaises » tuiles. L'union des bonnes tuiles permet donc de synthétiser des valeurs de paramètres correspondant au bon comportement du système.

Notons enfin qu'il est tout à fait possible que la cartographie ne couvre pas l'intégralité de l'espace paramétrique dense au sein du domaine de départ ; il peut rester des « trous » entre deux points entiers successifs. En ce cas, il est possible de combler les trous restants en appelant de nouveau, de façon manuelle ou semi-automatisée, la méthode inverse sur un point (nécessairement non entier) à l'intérieur du trou. Notons néanmoins que cette heuristique peut elle-même ne pas suffire à couvrir l'intégralité de l'espace paramétrique dense au sein du domaine de départ. En effet, il peut y avoir une infinité de comportements différents dans un espace paramétrique borné. En ce cas, l'information fournie par la cartographie sera certes incomplète, mais néanmoins intéressante par rapport à des méthodes qui ne seraient tout simplement pas capables de terminer sur ces exemples.

Néanmoins, pour certaines classes de systèmes, nous montrons qu'il est possible de couvrir l'intégralité de l'espace paramétrique dense, non seulement au sein du domaine paramétrique de dé départ, mais également au-delà [AF10].

Avantages. Le principal avantage de notre méthode de cartographie est que la cartographie est en fait *indépendante* de la propriété que l'on cherche à vérifier. En effet, seule la *partition* entre bonnes et mauvaises tuiles dépend de la propriété. Par conséquent, si l'on cherche à vérifier une autre propriété que celle de départ, il est inutile de recalculer la cartographie. Il suffit d'effectuer la partition de nouveau, en testant la propriété sur un point dans chacune des tuiles.

Études de cas

IMITATOR. La méthode inverse a tout d'abord été implémentée dans le prototype IMITATOR [And09a], un script de 1500 lignes environ écrit en Python, et faisant appel à l'outil HYTECH [HHWT95].

Plusieurs expériences ont été réalisées, permettant de synthétiser des valeurs de paramètres garantissant un bon comportement pour un certain nombre d'études de cas [AEF09].

Malheureusement, IMITATOR souffrait de plusieurs limitations, notamment en raison de son interface avec HYTECH, ce qui limitait sérieusement ses performances. Par conséquent, une nouvelle implémentation d'IMITATOR a été décidée.

IMITATOR II. Un nouvel outil, IMITATOR II [And10a], a été réalisé afin d'implémenter la méthode inverse de façon plus efficace. IMITATOR II est désormais un outil d'un seul bloc d'environ 9000 lignes écrit en OCaml, et utilisant des bibliothèques de polyèdres. L'utilisateur a le choix entre la bibliothèque NewPolka, disponible au sein de la bibliothèque APRON [JM09], ou la bibliothèque *Parma Polyhedra Library* (PPL) [BHZ08]. Les ensembles de traces sont retournés sous forme graphique, grâce au module DOT du logiciel de visualisation graphique Graphviz [gWp]. La syntaxe, proche de celle de HYTECH avec quelques améliorations, est disponible dans le manuel utilisateur [And10b].

Outre la méthode inverse standard, une variante est implémentée, en l'occurrence celle avec point fixe plus lâche. L'utilisateur a donc le choix entre les deux algorithmes.

IMITATOR II implémente également la cartographie comportementale. Les ensembles de traces sont fournis sous forme graphique, et la cartographie elle-même est générée sous forme graphique pour deux dimensions de l'espace des paramètres. La méthode inverse n'est pas appelée sur chacun des points entiers du domaine paramétrique de départ, mais uniquement sur ceux qui ne sont pas encore couverts par une tuile. La partition en bonnes et mauvaises tuiles est en revanche effectuée manuellement dans la version courante de l'outil. Une version automatisée, en faisant appel à un outil externe, par exemple UPPAAL, est en cours de réflexion.

Plusieurs options sont disponibles, notamment des options permettant de limiter le nombre d'itérations ou le temps d'exécution de l'algorithme, ce qui autorise l'obtention de résultats partiels pour des exemples dont l'analyse ne termine pas en général.

Grâce à la réécriture de l'outil, et diverses optimisations de l'algorithme, le temps d'exécution de la méthode inverse dans IMITATOR II a drastiquement diminué par rapport à la première version d'IMITATOR, permettant un gain en temps d'exécution d'un facteur 10 à 1000 selon les exemples.

Études de cas. Un certain nombre d'exemples de la littérature, et d'études de cas réelles, ont été étudiés, en particulier des circuits asynchrones, et des protocoles de communication [And10c]. L'application de la méthode inverse ou de la méthode de cartographie nous a permis d'optimiser des bornes temporelles dans des circuits mémoires, de synthétiser des valeurs de bon fonctionnement, ou de donner des conditions de robustesses pour un certain nombre d'exemples.

En particulier, nous avons étudié les protocoles de communication CSMA/CD [KNSW07], RCP [KNS03] et BRP [DKRT97]. En ce qui concerne les circuits asynchrones, nous avons étudié le comportement de plusieurs exemples de circuits décrits dans la littérature [CC05, CC07], mais également une abstraction de la mémoire SPSMALL [CEFX09] commercialisée par le fabricant de composants électroniques ST-Microelectronics, ce qui a permis une optimisation des temps de maintien des signaux d'entrée.

Enfin, nous avons défini des plages de bon fonctionnement pour les paramètres temporels d'une architecture d'automatisation en réseau [AAC+09], et effectué une comparaison avec une technique dichotomique consistant à tester un grand nombre de valeurs entières des paramètres vis-à-vis d'une propriété donnée, à l'aide de l'outil UPPAAL. Si la zone générée par méthode dichotomique était nettement plus large que la zone générée par IMITATOR, celle-là était seulement en trois dimensions, alors que celle-ci était dense, et en 7 dimensions.

Extension aux systèmes probabilistes

Nous décrivons ici une application de la méthode inverse aux systèmes probabilistes.

Automates temporisés probabilistes. Nous considérons ici le modèle des *automates temporisés probabilistes* [KNSS02], où les actions discrètes des automates probabilistes sont remplacées par des *distributions* d'actions. En d'autres termes, dans un état de contrôle donné, pour une action donnée, il est désormais possible d'atteindre différents états de contrôles avec différentes probabilités. Ce formalisme est intéressant pour vérifier des systèmes probabilistes, tels que des protocoles avec une part d'aléatoire, ou des systèmes tolérants aux fautes.

La sémantique des automates temporisés probabilistes s'entend pour un *ordonnanceur* donné, c'est-à-dire une fonction qui résout le non-déterminisme, et associe à chaque état de contrôle *une* action de sortie. Pour cette action, il existe bien entendu plusieurs transitions probabilistes. Pour une ordonnanceur donné, la sémantique est alors proche de celle des automates temporisés, et s'entend désormais en terme d'exécutions probabilistes, ou successions d'états concrets et de transitions probabilistes, c'est-à-dire étiquetées par une probabilité. Une trace probabiliste est une exécution probabiliste dont on abstrait la valeur des horloges, c'est-à-dire une succession d'états de contrôles et de transitions probabilistes. Il est alors possible définir la probabilité minimum ou maximum d'atteindre un état de contrôle donné : il s'agit de la probabilité minimum ou maximum pour tous les ordonnanceurs possibles et pour toutes les exécutions possibles.

L'outil de vérification probabiliste PRISM [HKNP06] permet notamment le calcul de telles probabilités.

Automates temporisés probabilistes paramétrés. Comme pour les automates temporisés, le comportement des automates temporisés probabilistes est sensible aux valeurs des constantes utilisées dans les gardes et invariants. Par conséquent, il est intéressant de raisonner paramétriquement. Nous définissons donc le modèle des automates temporisés probabilistes paramétrés [AFS09], dans la lignée de la paramétrisation des automates temporisés en automates temporisés paramétrés : nous autorisons dans les gardes et invariants l'usage de *paramètres* en lieu et place des constantes rationnelles.

Ce modèle reste en revanche purement syntaxique, et ne sert qu'à décrire de façon paramétrique le système que l'on souhaite vérifier. Le calcul des probabilités minimum ou maximum d'accessibilité s'effectuera systématiquement pour des modèles instanciés de ces automates temporisés probabilistes paramétrés, c'est-à-dire des automates temporisés probabilistes classiques.

Motivation. La vérification du comportement d'un système temporisé probabiliste peut s'effectuer grâce à l'outil PRISM. Néanmoins, cet outil est extrêmement sensible à la taille des constantes utilisées dans le modèle. En effet, PRISM utilise un modèle à temps discret, et le fait que le système reste n unités de temps dans un état de contrôle est modélisé par n itérations. Or, en cas de parallélisme, une explosion du nombre d'états survient rapidement, et l'outil n'est plus à même de calculer les probabilités – ou alors après un très important temps de calcul. Par conséquent, il est intéressant d'utiliser des constantes les plus petites possibles telles que les valeurs des probabilités minimum ou maximum d'accessibilité soient les mêmes. Nous décrivons dans ce qui suit comment la méthode inverse permet cette réduction des constantes.

Extension de la méthode inverse. Soit un automate temporisé probabiliste paramétré et une instance de référence des paramètres. Appliquons la méthode inverse développée précédemment à une version *non-probabiliste* de l'automate temporisé probabiliste paramétré, c'est-à-dire un modèle dont on remplace les transitions probabilistes par un simple non-déterminisme. Alors, pour toute instance vérifiant la contrainte synthétisée, la valeur des probabilités minimum et maximum d'accessibilité dans l'automate temporisé probabiliste paramétré sous cette instance est la même [AFS09].

Il suffit alors, pour diminuer la valeurs des constantes du système, de choisir une instance « suffisamment petite » dans la contrainte, et d'appliquer PRISM sur le modèle avec ces constantes réduites. Expérimentalement, les contraintes synthétisées par IMITATOR II ont permis de drastiquement réduire le temps de calcul des probabilités pour le protocole CMSA/CD [KNSW07], et ont permis de calculer des probabilités pour le *Root Contention Protocol* [HRSV02] et pour le protocole IEEE 802.11 *(Wireless Local Area Network Protocol)* [pWpb, KNS02], alors que la taille importante des constantes originales ne permettait tout simplement pas de les calculer en utilisant PRISM.

Avantages. Outre l'avantage majeur de la réduction des constantes, la synthèse de contraintes par la méthode inverse dans le cadre probabiliste permet, comme dans le cas non-probabiliste, une garantie de *robustesse*. En outre, elle permet d'éviter un grand nombre de calculs lorsque l'on cherche à vérifier des probabilités pour de nombreuses valeurs des paramètres temporels.

Cartographie probabiliste. L'extension de la cartographie comportementale au cas probabiliste est dès lors immédiate. Les probabilités minimum et maximum d'accessibilité sont homogènes dans chaque tuile. Là où la cartographie comportementale permettait une partition binaire en bonnes et mauvaises tuiles, la cartographie probabiliste permet, pour une probabilité minimum ou maximum donnée, une partition *quantitative* : à chaque tuile correspond une valeur de la probabilité. À notre connaissance, aucune autre technique ne permet de garantir la préservation des valeurs de probabilités dans un cadre probabiliste temporisé.

Une méthode inverse pour les graphes valués

Nous proposons ici une adaptation de la méthode inverse définie dans le cadre des automates temporisés à deux autres types de modèles : l'algorithme de Floyd–Warshall déterminant le plus court chemin dans un graphe orienté valué, et l'algorithme d'itération de politiques dans le cadre des processus de décision markoviens.

Graphes orientés valués. Nous nous intéressons dans cette section à l'algorithme de Floyd–Warshall [Flo62], qui calcule le plus court chemin entre tous les couples de sommets d'un graphe orienté valué. Un graphe orienté valué est un ensemble de sommets, et un ensemble de transitions étiquetées par des coûts.

Un problème classique de la littérature est de calculer le plus court chemin, c'est-à-dire le chemin de coût minimal, entre tous couples de sommets du graphe. Le problème auquel on s'intéresse est le suivant : pour un graphe orienté valué, peut-on changer certains des coûts sans affecter les plus courts chemins ?

Pour résoudre ce problème, on considère alors un graphe orienté valué *paramétré*, c'est-à-dire un graphe orienté valué où les coûts ne sont plus des constantes, mais des *paramètres*, ou constantes rationnelles inconnues.

Nous proposons une adaptation de l'algorithme de Floyd–Warshall, qui permet de générer des contraintes *garantissant* que le plus court chemin reste le plus court chemin, pour toute instance vérifiant cette contrainte [And09b]. Cet algorithme a été implémenté sous la forme de l'outil INSPEQTOR (pour *INference of Shortest Paths with EQuivalent Time-abstract behaviOR*), un programme OCaml de 3000 lignes.

Une application immédiate de cette méthode est en fait l'étude de systèmes temporisés tels que les circuits. Dans ce cas, les composants (et câbles) d'un circuit sont considérés comme des éléments que l'électricité mettra un certain temps à traverser, certes faible, mais non négligeable dans le cadre de la vérification de tels systèmes. Dès lors, chaque composant du système peut être représenté par l'arc d'un graphe orienté valué. Il est alors intéressant de savoir si l'on peut changer un composant du système par un autre plus lent, dans un souci économique, et ce sans remettre en cause le fonctionnement global du système. En générant une contrainte grâce à cette adaptation de la méthode inverse, il est alors possible de changer certains composants du système, pour peu que leur nouvelle durée de traversée vérifie toujours la contrainte.

Processus de décision markoviens. Nous considérons ici une extension de la méthode inverse aux processus de décision markoviens. Un *processus de décision markovien* [Bel57] est un graphe orienté valué auquel on ajoute des actions sur les arcs. En outre, chaque arc porte une probabilité telle que, pour un sommet du graphe et une action donnés, la somme des probabilités quittant ce sommet via cette action soit égale à 1.

Un problème classique des processus de décision markoviens est de déterminer une *politique optimale*, c'est-à-dire de résoudre le non-déterminisme en choisissant préalablement une action de sortie pour chaque sommet, et ce afin de minimiser le coût global du graphe. Un processus de décision markovien dont le non-déterminisme a été résolu devient alors une *chaîne de Markov* [KMST59], modélisable par exemple grâce à l'outil PRISM [HKNP06]. Une réponse classique à ce problème est d'utiliser un *algorithme d'itération de politiques* [How60]. Cet algorithme part d'une politique quelconque puis, à chaque itération, calcule la valeur associée à chaque sommet pour cette politique grâce à un algorithme classique dit *d'itération de valeurs*, et améliore la politique en conséquence, et ce jusqu'à ce que celle-ci soit optimale.

Le problème que l'on cherche à résoudre ici est le suivant : pour un processus de décision markovien donné, peut-on changer certains des coûts sans affecter la politique optimale ? Suivant un raisonnement similaire à celui de la méthode inverse définie dans le cadre des automates temporisés ou des graphes orientés valués, nous considérons des processus de décision markoviens *paramétrés*, c'est-à-dire où les coûts sont remplacés par des paramètres.

Nous définissons une adaptation de l'algorithme d'itération de politiques, qui prend en entrée un processus de décision markovien paramétré et une instance de référence des paramètres [AF09]. Il synthétise une contrainte sur les paramètres telle que, pour toute instance des paramètres vérifiant cette contrainte, la politique optimale du processus de décision markovien sous cette instance sera identique à la politique optimale du processus de décision markovien sous l'instance de référence.

Cet algorithme a été implémenté sous la forme de l'outil IMPRATOR (pour *Inverse Method for Policy with Reward AbstracT behaviOR*), un programme OCaml de 4300 lignes. Pour un système avec 11 états et 132 transitions, correspondant à la modélisation du parcours d'un robot dans un espace physique borné, une contrainte est générée en 0,17 seconde par IMPRATOR.

Remarques. Nous avons présenté ici une adaptation de la méthode inverse à deux algorithmes de la littérature traitant des plus courts chemins dans un cadre probabiliste ou non. Nous travaillons en ce moment sur une adaptation à d'autres algorithmes.

En particulier, l'adaptation à un algorithme d'itération de politiques pour les processus de décision markoviens à deux coûts est à l'étude. Cette présence de deux coûts distincts permet ainsi de modéliser la gestion de puissance dynamique (*dynamic power management* [PBBDM98]) des systèmes temps réel, où l'on souhaite par exemple minimiser l'énergie tout en gardant les pertes de requêtes inférieures à une certaine valeur.

Conclusion et perspectives

Conclusions. Nous avons présenté dans cette thèse des techniques pour la synthèse de constantes temporelles dans les systèmes temps réel. L'algorithme de la méthode inverse permet la synthèse de paramètres dans le cadre des automates temporisés, en garantissant le même comportement en termes d'ensembles de traces que sous une instance de référence donnée. Ceci permet la préservation de propriétés temporelles linéaires, garantit une robustesse du système, et permet l'optimisation de certaines constantes temporelles.

L'algorithme de cartographie comportementale permet de réaliser une cartographie des automates temporisés, en partitionnant tout ou partie de l'espace paramétrique en tuiles comportementales pour lesquelles les ensembles de traces sont homogènes. Cette cartographie est indépendante de la propriété que l'on cherche à vérifier : seule la partition en bonnes et mauvaises tuiles, permettant de synthétiser la contrainte correspondant au bon comportement, dépend de la propriété.

Ces deux algorithmes ont été implémentés dans IMITATOR II, un outil qui nous a permis de synthétiser des contraintes pour un certain nombre d'études de cas, circuits asynchrones ou protocoles de communication.

Nous avons étendu la méthode inverse et la méthode de cartographie comportementale aux automates temporisés probabilistes, garantissant ainsi les mêmes valeurs de probabilités minimum et maximum d'accessibilité dans chaque tuile. Ceci permet notamment la réduction des constantes, dont dépendent fortement les performances de l'outil PRISM.

Enfin, nous avons présenté ici une adaptation de la méthode inverse à deux algorithmes de la littérature traitant des plus courts chemins dans un cadre probabiliste ou non. Deux prototypes, INSPEQTOR et IMPRATOR ont été implémentés.

Perspectives. Il serait intéressant d'étendre la méthode inverse au formalisme des *automates hybrides*, où les horloges peuvent évoluer avec des vitesses différentes.

Plutôt que de considérer une méthode basée sur une instance de référence, il serait également intéressant d'envisager une *trace* de référence. Bien que cela remette fondamentalement en cause les bases de la méthode inverse, cela présenterait l'avantage majeur d'autoriser de considérer les ordres partiels entre actions indépendantes. Des travaux proches pourraient également concerner la synthèse de paramètres afin de garantir la correction d'une formule de logique temporelle.

Quant à l'outil IMITATOR II, de nombreuses améliorations pourraient être effectuées à plus ou moins long terme. Parmi celles-ci, la partition automatique entre bonnes et mauvaises tuiles pourrait être réalisée à l'aide d'un outil externe, par exemple UPPAAL. L'implémentation de la seconde variante avec union des contraintes serait également intéressante en pratique. La classe des automates temporisés autorisés par l'outil pourrait être étendue, notamment aux actions urgentes et états de contrôles urgents. Enfin, il pourrait être intéressant de considérer une cartographie *dynamique*, où les unités entre les points à sélectionner (pour l'heure les entiers) serait raffinée automatiquement afin de remplir de possibles « trous ».

Finalement, nous avons montré par des variantes de la méthode inverse définies dans deux autres formalismes, en l'occurrence les graphes orientés valués et les processus de décision markoviens, que la méthode inverse n'était en rien intrinsèquement liée aux automates temporisés. Une extension aux automates à coûts [ATP01, BFH$^+$01], ou aux réseaux de Petri temporels [Mer74] pourrait notamment être envisagée.