# Constraint modelling and solving of some verification problems

Anicet Bart

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Thèse de Doctorat

Anicet BART

*Mémoire présenté en vue de l'obtention du*
***grade de Docteur de l'École nationale supérieure Mines-Télécom Atlantique Bretagne-Pays de la Loire***
*sous le sceau de l'Université Bretagne Loire*

**École doctorale : Mathématiques et STIC (MathSTIC)**

**Discipline : Informatique, section CNU 27**
**Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)**

**Soutenue le 17 octobre 2017**
**Thèse n° : 2017IMTA0031**

# Constraint Modelling and Solving of some Verification Problems

## JURY

| | |
|---|---|
| Présidente : | **M^me^ Béatrice BÉRARD**, Professeur, Université Pierre et Marie Curie, Paris-VI |
| Rapporteurs : | **M. Salvador ABREU**, Professeur, Universidade de Évora, Portugal |
| | **M^me^ Béatrice BÉRARD**, Professeur, Université Pierre et Marie Curie, Paris-VI |
| Examinateurs : | **M. Nicolas BELDICEANU**, Professeur, Institut Mines-Télécom Atlantique, Nantes |
| | **M. Philippe CODOGNET**, Professeur, Université Pierre et Marie Curie, Paris-VI |
| Invité : | **M. Benoît DELAHAYE**, Maître de conférence, Université de Nantes |
| Directeur de thèse : | **M. Éric MONFROY**, Professeur, Université de Nantes |
| Co-encadrante de thèse : | **M^me^ Charlotte TRUCHET**, Maître de conférence, Université de Nantes |

# Constraint Modelling and Solving of some Verification Problems

Mémoire présenté en vue de l'obtention du grade de

**Docteur de l'École nationale supérieure Mines-Télécom Atlantique Bretagne-Pays de la Loire - IMT Atlantique**

sous le sceau de l'Université Bretagne Loire

Anicet BART

**École doctorale :** Sciences et technologies de l'information, et mathématiques
**Discipline :** Informatique et applications, section CNU 27
**Unité de recherche :** Laboratoire des Sciences du Numérique de Nantes (LS2N - UMR 6004)

**Directeur de thèse :** Pr. Éric MONFROY, professeur de l'Université de Nantes
**Co-encadrante :** Dr. Charlotte TRUCHET, maître de conférence de l'Université de Nantes

# Jury

Directeur de thèse :

**M. Éric MONFROY**, *Professeur*, Université de Nantes

Co-encadrante :

**Mme Charlotte TRUCHET**, *Maître de conférence*, Université de Nantes

Rapporteurs :

**M. Salvador ABREU**, *Professeur*, Université d'Évora, Évora (Portugal)
**Mme Béatrice BÉRARD**, *Professeur*, Université Pierre et Marie Curie, Paris-VI

Examinateurs :

**M. Nicolas BELDICEANU**, *Professeur*, Institut Mines-Télécom Atlantique, Nantes
**M. Philippe CODOGNET**, *Professeur*, Université Pierre et Marie Curie, Paris-VI

Invité :

**M. Benoît DELAHAYE**, *Maître de conférence*, Université de Nantes

# Constraint Modelling and Solving of some Verification Problems

**Short abstract:** Constraint programming offers efficient languages and tools for solving combinatorial and computationally hard problems such as the ones proposed in program verification. In this thesis, we tackle two families of program verification problems using constraint programming. In both contexts, we first propose a formal evaluation of our contributions before realizing some experiments. The first contribution is about a synchronous reactive language, represented by a block-diagram algebra. Such programs operate on infinite streams and model real-time processes. We propose a constraint model together with a new global constraint. Our new filtering algorithm is inspired from Abstract Interpretation. It computes over-approximations of the infinite stream values computed by the block-diagrams. We evaluated our verification process on the FAUST language (a language for processing real-time audio streams) and we tested it on examples from the FAUST standard library. The second contribution considers probabilistic processes represented by Parametric Interval Markov Chains, a specification formalism that extends Markov Chains. We propose constraint models for checking qualitative and quantitative reachability properties. Our models for the qualitative case improve the state of the art models, while for the quantitative case our models are the first ones. We implemented and evaluated our verification constraint models as mixed integer linear programs and satisfiability modulo theory programs. Experiments have been realized on a PRISM based benchmark.

**Keywords:** Constraint Modelling - Constraint Solving - Program Verification - Abstract Interpretation - Model Checking

# Modélisation et résolution par contraintes de problèmes de vérification

**Résumé court :** La programmation par contraintes offre des langages et des outils permettant de résoudre des problèmes à forte combinatoire et à la complexité élevée tels que ceux qui existent en vérification de programmes. Dans cette thèse nous résolvons deux familles de problèmes de la vérification de programmes. Dans chaque cas de figure nous commençons par une étude formelle du problème avant de proposer des modèles en contraintes puis de réaliser des expérimentations. La première contribution concerne un langage réactif synchrone représentable par une algèbre de diagramme de blocs. Les programmes utilisent des flux infinis et modélisent des systèmes temps réel. Nous proposons un modèle en contraintes muni d'une nouvelle contrainte globale ainsi que ses algorithmes de filtrage inspirés de l'interprétation abstraite. Cette contrainte permet de calculer des sur-approximations des valeurs des flux des diagrammes de blocs. Nous évaluons notre processus de vérification sur le langage FAUST, qui est un langage dédié à la génération de flux audio. La seconde contribution concerne les systèmes probabilistes représentés par des chaînes de Markov à intervalles paramétrés, un formalisme de spécification qui étend les chaînes de Markov. Nous proposons des modèles en contraintes pour vérifier des propriétés qualitatives et quantitatives. Nos modèles dans le cas qualitatif améliorent l'état de l'art tandis que ceux dans le cas quantitatif sont les premiers proposés à ce jour. Nous avons implémenté nos modèles en contraintes en problèmes de programmation linéaire en nombres entiers et en problèmes de satisfaction modulo des théories. Les expériences sont réalisées à partir d'un jeu d'essais de la bibliothèque PRISM.

**Mots clés :** modélisation par contraintes - résolution par contraintes - vérification de programmes - interprétation abstraite - vérification de modèles

# Contents

Chapter 1

# Introduction

## Contents

Computer scientists started to write programs in order to produce softwares realizing dedicated tasks faster and more efficiently than a human could perform. However, in ad-hoc developments the more complex is the problem to solve the longer it takes to write its corresponding solving program. Moreover, few modifications in the description of the problem to solve may impact many changes in the program. The artificial intelligence research domain tries to develop more generic approaches such that a single artificially intelligent program may solve a wide variety of heterogenous problems. Constraint programming is a research axis in the artificial intelligence community where constraints are sets of rules to be satisfied and the intelligent program must find a solution according to these rules. Thus, the objective of the constraint programming community is to produce languages and tools for solving constraint based problems. Such problems are expressed in a declarative manner where programs consist in a set of rules (called constraints) to be satisfied. Thus, a constraint programming user enumerates his/her rules and uses a *black-box tool* (called solver) for solving his/her problem. These are two major research activities in constraint programming: modelling and solving. The modelling activity works on the expressiveness of the constraint language and manipulates constraint programs in order to improve the resolution process. The solving activity consists in developing algorithms, tools, and solvers for improving the efficiency of the resolution process.

For the last decades computers and information systems have been highly democratized for private and company usages. In both contexts, more and more complex systems are developed in order to realize a wide variety of applications (smart applications, embedded systems for air planes, medical robot assistants, etc.). As for many other production fields, writing systems must respect *quality* rules such as conformity, efficiency, and robustness. In this thesis, we are concerned by the *verification* problem consisting in verifying if an application, a program, a system matches its *specifications* (*i.e.*, its expected behavior). This concern gained an important interest after social or business impacts are identified, or after past failures. One of the most remarkable examples is the crash of the Ariane 5 missile, 36 seconds after its launch on June 4, 1996. The accident was due to a conversion of a 64-bit floating point number into a 16-bit integer value. Another example is the bug in Intel's Pentium II floating-point division unit in the 90's, which forced to replace faulty processors, severely damaged Intel's reputation, and implied a loss of about 475 million US dollars. These events happened in the 90', and software are now more and more used to automatically control critical systems such as nuclear power plants, chemical plants, traffic control systems, and storm surge barriers. Furthermore, even programs with less critical impact require attention, since the competition between products gives benefits to the systems with less bugs, a better reactivity, etc. Thus, the verification objective is to attest the validity of a system according to an expected behavior.

While such problems may be solved using add-hoc techniques or proper tools, it appears that by nature or by reformulation of the problem, constraint programming offers

effective solutions. For instance, the system/program can be formulated as a set of rules and the expected behavior as a set of constraints. Thus, verifying the validity of the system/program behavior consists in determining if satisfying the rules implies to satisfy the expected behavior. On the other hand, some verification considerations may produce combinatorial problems. In this context constraint programming clearly appears as a suitable solution. In this thesis we tackle *program verification problems* as applications to be treated using constraint programming.

## 1.1 Scientific Context

As said before this thesis concerns constraint programming modelling and solving for some program verification problems. In this section we briefly present all the scientific context of the thesis by identifying separately the various scientific thematics tackled in this manuscript. We start by presenting constraint modelling and solving. Then, we continue with the two program verification approaches used in this thesis, and we conclude by presenting the two programming paradigms to be verified in this thesis.

**Constraint Modelling.** A Constraint Satisfaction Program (CSP for short) is a set of *constraints* over *variables* each one associated with a *domain*. Thus, constraint *modelling* consists in formulating a given problem into a CSP. There exist various research communities each one dedicated to model families of CSPs. Recall first that the general problem of satisfying a CSP (*i.e.*, finding a valuation of the variables satisfying all the constraints in the CSP) is a hard problem. There exists CSP families being tractable in exponential, polynomial, or ever linear time. In this thesis we consider constraint modelling ranging from mathematical programming such as continuous and mixed integer linear programing (respectively `LP` and `MILP` for short), finite and continuous domains programs without linearity restrictions on the constraints (respectively named `FD` and `CD` for short), and Satisfiability Modulo Theory (`SMT` for short) mixing Boolean and theories such as arithmetics. See Section 2.2 for more details.

**Constraint Solving.** Various tools, named *solvers*, have been developed for solving CSP instances. Each one is mainly specialized to solve specific CSP families (*e.g.*, unbounded integer linear arithmetics, constraints over variables with finite domain, continuous constraints). The combinatorics implied by the relations between the variables in the constraints makes a CSP hard to solve. This requires to explore *search spaces* composed of all the valuations possibly candidate for solving the problem. However, the size of such search space is exponential in terms of the problem sizes (number of variables) in general. Huge research efforts has been put into solvers in order to propose tools for (intelligently) explore huge search spaces and solve CSP instances. See Section 2.3 for more details.

**Program Verification** A program describes the behavior of a possibly infinite process by defining possible transitions from states to states. Due either to the runtime environment or to the non determinism of the state successions, one program may have a finite or even an infinite number of possible runs. Also, according to the nature of a program its runs may encounter either a finite or an infinite number of states in theory. Thus, program verification consists in determining if the program traces (*i.e.*, the state sequences realized by the runs) respect a given property. These properties may be time dependent (*e.g.*, for each run the state $A$ must be encountered after the state $B$, the state $A$ must not be encountered before a given time $t$) or time independent (*e.g.*, for each run all the variables are bounded by given constants). There are two main approaches for verifying properties on program: *dynamic analysis* vs. *static analysis*. Dynamic analysis requires to run the program to attest the validity of the property. On the contrary, static analysis performs verification at compilation time without running the program/system (roughly speaking dynamic analysis can be considered as an online process compared to static analysis which is an offline process). See Section 3.1 for more details. In this thesis we only consider complete static analyzes of programs with infinite runs (*i.e.*, we do not consider dynamic and bounded analyzes).

**Abstract Interpretation.** Abstract Interpretation is a program verification technique for static program analysis. In this context, we consider programs with unbounded running times and infinite state systems. Recall that in such cases the general program verification problem is undecidable since this class of problems contains the halting problem. Thus, Abstract Interpretation provides a verification process, which terminates, using over-approximations of the semantics of the program to verify. Indeed, well chosen abstractions produce semi-decidable problems. Thus, verification tools based on abstract interpretation either prove the validity of the property or may not conclude. Hence, such method cannot find counter-examples and falsify properties. See Section 3.2 for more details.

**Model Checking.** Model Checking is a program verification technique for static program analysis. As for Abstract Interpretation, programs/models to be verified may have unbounded running times and infinite state space. Thus, model checking is a verification method that explores all possible system states. In this way, it can be shown that a given system model truly satisfies a certain property. Hence, such method proves the validity or the non validity of the property. More specifically, it can return a counter-example in non validity case. See Section 3.3 for more details.

**Synchronous Reactive Language.** Motivated by the nature of embedded controllers requiring to be reactive to the environment at real-time, synchronous languages have been designed for programming reactive control systems. These languages naturally deal with

the complexity of parallel systems. Indeed, parallel computations are realized in a lock-step such that all computations are synchronized reactions. Hence, this synchronization ensures by construction a guarantee of determinism and deadlock freedom. Finally, these languages abstract away all architectural and hardware issues of embedded, distributed systems such that the programmer can only concentrate on the functionalities. Instances of such languages are Faust, Lustre, and Esterel and have been successfully used in the context of critical systems requiring strong verification (*e.g.*, space applications, railway, and avionics) using certified compiler (*e.g.*, Scade [Sca] tool from Esterel Technologies providing a DO-178B level A certified compiler). Chapter 4 concerns the verification of synchronous reactive languages.

**Probabilistic Programming Language.** Various systems are subject to phenomena of a stochastic nature, such as randomness, message loss, probabilistic behavior. Probabilistic programming languages are used to describe such systems using probabilities to define the sequence of states in the program. One of the most popular probabilistic models for representing stochastic behaviors are the *discrete-time Markov Chains* (MCs for short). Instance of probabilistic programming languages for writing MCs are Problog and Prism. Chapter 5 concerns the verification of models extending the Markov chain model describing parametrized probabilistic systems.

## 1.2 Problems and Objectives

As presented in the previous sections, program verification is a computationally hard problem with major issues. Recall first that a program describes the behavior of a possibly infinite process by defining possible transitions from states to states. However, the verification is performed on an abstraction of the program named model[1]. In this thesis, we consider finite models with infinite state spaces and infinite runs.

Even if a program admits a priori an infinite state space its executions may encounter a (potentially infinite) subset of the declared state space. Thus, one would like to determine this smaller state space in order to verify the non reachability of undesired states. This problem is reducible to the search of program over-approximations, *i.e.*, bounding all the program variables. This is an objective of Abstract Interpretation where the program describing precisely the system evolution from a state to another, named the concrete program, is abstracted. This abstracted construction is related to the concrete program in such a manner that if an over-approximation holds for the abstraction then, this approximation also holds for the concrete program. Furthermore, constraint programs allow to describe over-approximations such as convex polyhedrons using linear

[1]We already introduced the word *model* in the constraint programming context. The reader must be careful that this word has an important role in both contexts.

constraints, ellipsoids using quadratic constraints, etc. Thus, since constraint programming is a generic declarative programing paradigm it may be seen as a verification process for over-approximating variable in declarative programs. In the first contribution, we consider a block-diagram language where executions are infinite streams and the objective is to bound the stream values using constraint programming.

However, bounding the state space is not enough for some verification requirements. In our second problem, the objective is to determine if a specific state is reachable at execution time. Indeed, abstractions can only determine if a specific state is unreachable. For this verification problem, we consider programs representable as finite graph structures where the nodes form the state space and the edges give state to state transitions. Thus, verifying the reachability of a state in such a structure is performed by activating or deactivating transitions in order to reach the target state. However, these activations can be restricted by guards, or other structural dependent rules. Clearly, this corresponds to a combinatoric problem to solve. For this reason, since one of the objectives of constraint programming is to solve highly combinatorial problems, the verification community is interested in the `CP` tools.

Some links between constraint programming and program verification are presented in Section 3.4. To conclude, constraint programming proposes languages to model and solve problems by focusing on the problem formulation instead of the resolution process. Program verification leads to problems which by definition or by nature are close to constraint programs. Thus, the verification community uses constraint programming tools for developing analyzers instead of producing ad-hoc algorithms. In this thesis, we position ourself as constraint programmers and we consider verification problems as applications. Thus, our objective is to present how the constraint programming advances in modelling and solving helps to answer some verification problems.

## 1.3 Contributions

The contributions are split into two distinct chapters, and they are related to different verification research axes, but both using constraint programming. The first contribution applies constraint programming to verify some properties of a real-time language, while the second one is about verification of extensions of Markov chains. Here are the abstracts of these two contributions.

**Verifying a Synchronous Reactive Language with constraints (Chapter 4).** Formal verification of real time programs in which variables can change values at every time step, is difficult due to the analyses of loops with time lags. In our first contribution, we propose a constraint programming model together with a global constraint and a filtering algorithm for computing over-approximation of real-time streams. The global

constraint handles the loop analysis by providing an interval over-approximation of the loop invariant. We apply our method to the FAUST language which is a language for processing real-time audio streams. We conclude with experiments that show that our approach provides accurate results in short computing times. This contribution has been published in a national conference [1], an international conference [2], and a journal [3].

**Verifying a Parametric Probabilistic Language with constraints (Chapter 5).** Parametric Interval Markov Chains (pIMCs) are a specification formalism that extends Markov Chains (MCs) and Interval Markov Chains (IMCs) by taking into account imprecision in the transition probability values: transitions in pIMCs are labeled with parametric intervals of probabilities. In this work, we study the difference between pIMCs and other Markov Chain abstractions models and investigate the three usual semantics for IMCs: once-and-for-all, interval-Markov-decision-process, and at-every-step. In particular, we prove that all these semantics agree on the maximal/minimal reachability probabilities of a given IMC. We then investigate solutions to several parameter synthesis problems in the context of pIMCs – consistency, qualitative reachability, and quantitative reachability – that rely on constraint encodings. Finally, we conclude with experiments by implementing our constraint encodings with promising results. This contribution has been published in a national conference [4], an international workshop without proceedings [5], and an international conference [6] (to appear).

## 1.4 Outline

The thesis in organized in four main chapters. Chapter 2 presents the constraint programming paradigm. Chapter 3 introduces program verification/model checking problems. We conclude this chapter by briefly introducing the two verification methods named Abstract Interpretation and Model Checking in order to motivate the two following chapters which respectively use these verification methods. Chapter 4 contains our first contribution. This chapter proposes a constraint programming model together with a global constraint and a filtering algorithm inspired from abstract interpretation for computing over-approximation of real-time streams. This chapter is also illustrated and validated by some experiments. Chapter 5 contains our second contribution. This chapter proposes constraint programming models for verifying qualitative and quantitative properties of parametric interval Markov chains with a model checking objective. This chapter also concludes with experiments. Note that both contribution chapters are self-contained including introduction, motivation, background, state of the art, contributions, and bibliography. Finally, Chapter 6 concludes this thesis document.

Chapter 2

# Constraint Programming

## Contents

## 2.1 Introduction

Computer scientists started to write programs in order to produce softwares realizing dedicated tasks faster and more efficiently than a human could perform. However, in ad-hoc developments the more complex is the problem to solve the longer it is to write its corresponding solving program. Moreover, few changes in the description of the problem to solve may impact many changes in the program. Thus, the artificial intelligence research domain tries to develop more generic approaches such that a single artificially intelligent program may solve a wide variety of heterogeneous problems. Among all possible artificial intelligences, we focus in this thesis on those dealing with constraint based problems. In such problems, one can enumerate a set of objects with possibly many different states for each object and a set of accepted configurations over these objects w.r.t. the states (cf. Definition 2.1.1).

**Definition 2.1.1** (Constraint Based Problem)**.** *Let $A$ be a set of objects, and $S$ be a set of object states. A* Constraint Based Problem *$\mathcal{P}$ over objects $A$ with states $S$ represents a set of* configurations *(*i.e.*, a set of associations between objects from $A$ and states in $S$). Formally $\mathcal{P} \equiv \mathcal{L}$ s.t. $\mathcal{L} \subseteq S^A$.*

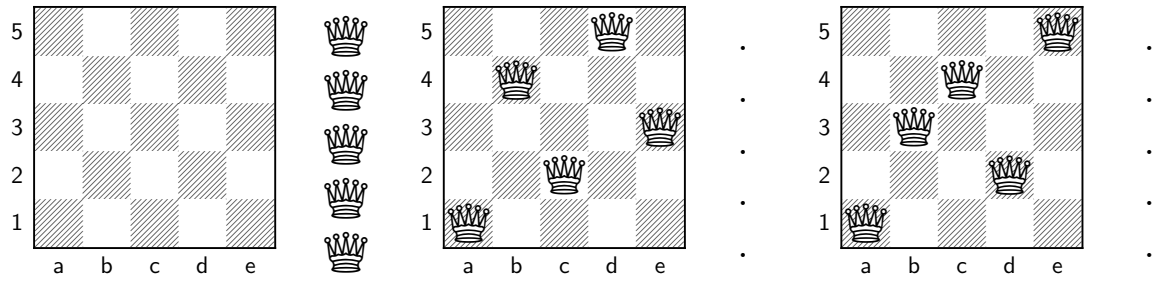
In this chapter, we first present constraint modelling (*i.e.*, variables, domains, constraints, and constraint programs definitions) and various research axes dedicated to constraint modelling (`SAT`, `CP`, `LP`, etc.). Then, we present these research axes dealing with constraint programs by describing their common resolution processes and their specific strategies developed in each one.

**Restrictions.** In this thesis we consider modelling with real, integer, and Boolean variables with finite or infinite domains without restrictions on the constraints (*e.g.*, enumerations, linear and non-linear inequations, Boolean compositions, global constraints) using the existential quantification of variables and being time-independent[1]. Finally, we consider complete methods for solving such models.

## 2.2 Constraint Modelling

Constraint modelling is the action of formulating a given constraint based problem into a constraint based program. Definition 2.1.1 recalled that a constraint based problem is described by a set of *objects*, a set of object *states*, and gathered into a set of *configurations*. Constraint programming uses a dedicated vocabulary. In the following we take care to well separate the constraint based problems from constraint based programs. Indeed constraint based problems are commonly expressed in a natural language while

[1]Note that Chapter 4 models a problem with time-dependency (verification of a reactive synchronous programming language). However the constraint modelling is time-independent.




(a) Empty 5×5 chessboard with 5 Queens.

(b) Queens positioning respecting "no threat" rules.

(c) Queens positioning violating twice the diagonal "no threat" rule.

Figure 2.1: 5-Queens problem illustrated with: (a) its 5×5 empty chessboard and its 5 queens; (b) a queen configuration satisfying the 5-Queens problem; and (c) a queen configuration violating the 5-Queens problem.

constraint programs are expressed in a mathematical (or mathematical-like) language or a programming language. A constraint program uses *variables* associated with *domains* linked by *constraints*. Roughly speaking, the variables with their domains will model the objects with their states in the constraint based program while the constraints will model the configurations in the constraint based program. We now present a wide landscape of variables, domains, and constraints encountered in constraint modelling while encoding a constraint based problem into a constrained program.

**Example 1** (n-Queens Problem)**.** The $n$-Queens problem will be our backbone example for illustrating constraint modelling and solving in this section. Let $n$ be a natural number. Thus we consider an $n \times n$ chessboard and $n$ queens. The $n$-Queens Problem objective is to place the $n$ queens on the chessboard such that no two queens threaten each other (*i.e.*, no two queens share the same row, column, or diagonal). In this example, objects are the $n$ queens and states are the $n \times n$ cells of the chessboard. Thus, a configuration is an arrangement of the $n$ queens on the chessboard.

### 2.2.1 Variables and Domains

A constraint based problem is described by a set of *variables*, each variable being associated with a non-empty set called its *domain*. From now on in this section, $X$ will refer to a set of $n$ variables $x_1, \ldots, x_n$, $D_x$ will be the domain associated to the variable $x \in X$, and $D$ will contain all the domains associated to the variables in $X$. We identify four variable types according to their domain. We say that a variable $x$ with domain $D_x$ is:

- a Boolean variable iff its domain is a binary set (*i.e.*, $D_x = \{\mathsf{true}, \mathsf{false}\}$)
- an integer variable iff its domain only contains integers (*i.e.*, $D_x \subseteq \mathbb{Z}$)
- a rational variable iff its domain only contains rational numbers (*i.e.*, $D_x \subseteq \mathbb{Q}$)

- a real variable iff its domain only contains real numbers (*i.e.*, $D_x \subseteq \mathbb{R}$)

A domain can be given in extension by enumerating all the elements composing it or in intension using an expression representing all its elements. One common compact representation is the *interval* representation together with the union of intervals. Formally, let $E \subseteq \mathbb{R}$ be a non-empty totally ordered set and $a, b \in \overline{\mathbb{R}}$[2] be two interval endpoints. We write $\mathbb{I}_E([a, b])$ for the set containing all the (closed, semi-opened, opened) intervals subsets of the interval $[a, b] \subseteq E$. When modelling, we usually separate real variables with interval domains from others. The first ones are called *continuous* variables while the remaining are called *discrete* variables. Furthermore, we separate *finite* variables (*i.e.*, variables whose domains have a finite number of elements) from *infinite* variables. For instance a finite variable can be introduced by domain enumeration (*e.g.*, domain $\{1, 2, \ldots, 50\}$) and infinite variables can be defined by interval domain (*e.g.*, domain $[-1, 1]$ subset of $\mathbb{R}$). Note that there exists other domains such that the symbolic domains where each domain may contain an infinite number of possibly ordered symbols, or the set domains where each domain element is a set of values. In this thesis we perform constraint modelling with Boolean, integer, rational, and real-number domains. Finally, a *valuation* of the variables in $X' \subseteq X$ is a mapping $v$ associating to each variable in $X'$ a value in its domain (*i.e.*, $v : X' \to D$ s.t. $v(x) \in D_x$ for all $x \in X'$).

**Example 2.** Here are some domain instances:

- $\{0, 1, \ldots, 100\} = [0, 100] \subseteq \mathbb{N}$ finite domain over integers
- $\{0, 1, 2, 3, 5, 7, 11\}$ finite discontinuous domain enumeration
- $[0, 100] \subseteq \mathbb{R}$ infinite continuous domain
- $\{0\} \cup [1, 100] \subseteq \mathbb{R}$ infinite semi-continuous domain

### 2.2.2 Constraints

A constraint is defined over a set of variables and represents a set of accepted valuations. Formally a constraint $c$ over the variables $X$ with domains $D$ is semantically equivalent to a set of valuations from $X$ to $D$: $c \equiv \mathcal{V}$ such that $\mathcal{V} \subseteq D^X$. Constraints can be represented in extension by enumerating accepted valuations or in intension by a predicate over the variables in the constraint. With Boolean variables, the atomic constraints are the logical predicates such as the negation ($\neg$), the conjunction ($\wedge$), the disjunction ($\vee$), the implication ($\Rightarrow$), the equivalence ($\Leftrightarrow$). For other domains, we consider atomic constraints as equations or inequations where their left-hand side and right-hand side are arithmetic expressions (*i.e.*, any mathematical expressions such that polynomials, trigonometric functions, logarithms, exponentials, etc.). In the context of finite variables, the Constraint Programming community proposes a catalogue of constraints with a high

[2] $\overline{\mathbb{R}}$ is the extended set of $\mathbb{R}$ and equals the union of $\mathbb{R}$ and its limits (*i.e.*, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$).

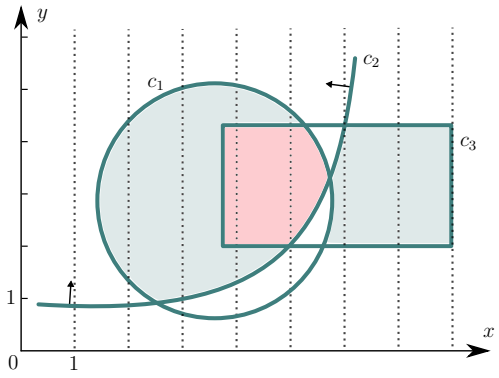



Figure 2.2: Three constraints $c_1$, $c_2$ and $c_3$ over two variables $x$ and $y$.

level semantics called *global constraints* (cf. Section 2.2.5). According to the domains considered in this thesis (*i.e.*, $\mathbb{B}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$) one important constraint characterization is the *linearity*. We say that a constraint is linear iff its arithmetic expressions are linear arithmetic expressions (*i.e.*, not containing products of variables). Less importantly one may also consider the convexity properties of the arithmetic expressions. Furthermore, recall that there exist two quantifiers: the existential and the universal quantifiers. Thus, in quantified constraints, variables are associated to quantifiers and the CSP is satisfiable iff the quantifiers hold for the given domains (*e.g.*, $\exists x \in [-1, 1], \forall y \in [0, 1] : x + y \leq 1$ is satisfiable). In this thesis we only consider constraints with the existential quantifier (*i.e.*, the universal quantifier is not allowed). Finally, a constraint problem is the composition of atomic constraints with logical operators.

**Example 3.** Figure 2.2 describes three constraints $c_1$, $c_2$, and $c_3$ over two variables $x$ and $y$. Geometrically speaking, constraint $c_1$ defines a disc, $c_2$ is an upper half-space, and $c_3$ is a rectangle. Thus, $c_1$ can be expressed by a quadratic inequality, $c_2$ by a non-linear inequality, and $c_3$ by a conjunction of four linear inequalities. The pink zone contains all the solutions of the CSP $\mathcal{C}_1$ with constraints $c_1$, $c_2$, $c_3$ over variables $x$, $y$ with respective domains $[0, 8] \subseteq \mathbb{R}$, and $[0, 6] \subseteq \mathbb{R}$. One may also consider the CSPs $\mathcal{C}_2$ and $\mathcal{C}_3$ which respectively contain the constraints $c_1 \wedge (c_2 \vee c_3)$ and $c_1 \Leftrightarrow (\neg c_2 \wedge c_3)$ producing different solution areas (*i.e.*, solution spaces/feasible regions).

### 2.2.3 Satisfaction and Optimization Problems

A constraint satisfaction problem consists in determining if a constraint satisfaction program (cf. Definition 2.2.1) is either *satisfiable* or *unsatisfiable*. Formally, a valuation $v$ satisfies a CSP $\mathcal{P} = (X, D, C)$ iff there exists a valuation $v$ over variables $X$ satisfying all the constraints in $C$ (*i.e.*, the set of constraints in $C$ is interpreted as a conjunction of constraints). If such a valuation $v$ exists we say that $\mathcal{P}$ is satisfiable (and $v$ is named a solution of $\mathcal{P}$), otherwise we say that $\mathcal{P}$ is unsatisfiable.

| | Real var. | Integer var. | Mixed var. | Finite var. |
|---|---|---|---|---|
| Linear | P | NP-complete | NP-complete | NP-complete |
| Non-linear | decidable | undecidable | undecidable | NP-complete |

Table 2.1: Complexity for the Constraint Satisfaction Problem Classes containing Linear and Non-Linear Constraints Problems over Real, Integer, Mixed, and Finite variables.

**Definition 2.2.1** (Constraint Satisfaction Program)**.** *A Constraint Satisfaction Program (*CSP *for short) is a triplet* $\mathcal{P} = (X, D, C)$ *where* $X$ *is a set of variables,* $D$ *contains the domains associated to the variables in* $X$*, and* $C$ *is a finite set of contraints over variables from* $X$*.*

In the following, we call CSP family a set of CSPs sharing properties (*e.g.*, only using integer variables, only considering linear constraints). Thus, according to a CSP family its theoretical complexity for the satisfaction problem may be polynomial or not, and either be undecidable. Table 2.1 from [7] synthesizes theoretical complexities for solving the satisfaction problem according to variables and constraints types. For instance the satisfaction of: a conjunction of linear constraints over real variables can be solved in polynomial time [8]; a conjunction of constraints over integer finite variables is an NP-complete problem [9]; non-linear constraints over unbounded integer variables is an undecidable problem [7].

### 2.2.4 Constraint Modelling

Given a problem to answer, the objective of constraint modelling is to encode the problem to be solved into a constraint program such that a solution of the constraint program can be translated into a solution of the original problem. Definition 2.2.2 recalls the concept of CSP modelling. Constraint modelling is presented in Definition 2.2.3. In order to construct a constrain program $\mathcal{P}'$ modelling a constraint based problem $\mathcal{P}$ one must find a correspondence relation linking the valuations satisfying $\mathcal{P}'$ with the configurations belonging to $\mathcal{P}$. Thus, if one is able to satisfy the CSP the correspondence relation ensures the existence of a configuration belonging to the constraint based problem. Furthermore, if the correspondence relation is decidable (ideally in polynomial time), one can construct at least one valid configuration from a solution given by the CSP.

**Definition 2.2.2** (Model)**.** *Let* $A$ *be a set of objects,* $S$ *be a set of object states, and* $\mathcal{P}$ *be a constraint based problem. We say that the* CSP $(X, D, C)$ *models* $\mathcal{P}$ *iff there exists a correspondence relation* $\mathcal{R} \subseteq D^X \times S^A$ *s.t.*

1. *for each* $(v, v') \in \mathcal{R}$*, the valuation* $v$ *satisfies* $C$ *and the configuration* $v'$ *belongs to* $\mathcal{P}$
2. *for each valuation* $v$ *satisfying* $C$ *there exists a configuration* $v'$ *s.t.* $(v, v') \in \mathcal{R}$

*3. for each configuration $v'$ in $\mathcal{P}$ there exists a valuation $v$ s.t. $(v, v') \in \mathcal{R}$*

**Definition 2.2.3** (Modelling)**.** *Let $\mathcal{L}$ be a set of constraint based problems. We say that* `M` *is a constraint modelling of $\mathcal{L}$ iff* `M` *is a mapping associating to each $\mathcal{P}$ in $\mathcal{L}$ a* CSP *$\mathcal{P}'$ s.t. $\mathcal{P}'$ models $\mathcal{P}$.*

Modelling a constraint based problem as a constraint satisfaction program can be characterized in four steps. Definition 2.2.3 requires the existence of a correspondence relation. Thus, the programmer mainly builds the CSP while taking into account the correspondence between the original problem and the developed CSP. Firstly, the programmer identifies the *decisions variables*: *i.e.*, the variables with a clear semantics in the problem to be solved. Secondly, she/he determines the *auxiliary variables* (*i.e.*, non decision variables used for intermediates constraints/computations). Thirdly, she/he sets the domain of each variable, also called the limits of each variable in the context of interval based domains. Fourthly, she/he adds the constraints that must be satisfied by the variables. These four steps are not necessarily straightforward and the programmer usually refines each step until a fix point is reached: the constructed CSP models the problem to solve. The following example proposes a modelling for the $n$-Queens problem. Note that this is a first modelling and that we are going to improve it in the following sections.

**Example 4** (Example 1 continued)**.** We propose a first CSP modelling $\mathtt{M}_0$ for solving the $n$-Queens problem where the decision variables model the columns and the lines chosen for the queens. Formally, let $\mathcal{L}$ be the set of all the $n$-Queens problems with $n \in \mathbb{N}$. $\mathtt{M}_0$ is the mapping associating to each $n$-Queens problem in $\mathcal{L}$ the CSP $(X, D, C)$ such that $X$ contains one variable $c_i$ and one variable $\ell_i$ per queen index $i \in \{1, \ldots, n\}$. These variables respectively indicate the column and the line position of the $i$th queen on the chessboard. Furthermore, the domain for all these variables is $\{1, \ldots, n\}$ and the constraints are the followings ones for each pair $(i, j)$ of two different queen indexes: 1. queens $i$ and $j$ are not on the same line: $\ell_i \neq \ell_j$; 2. queens $i$ and $j$ are not on the same column: $c_i \neq c_j$; 3. queens $i$ and $j$ are not on the same diagonal: $|(\ell_i - \ell_j)/(c_i - c_j)| \neq 1$. Note the abstraction difference between the modelling $\mathtt{M}_0$ and the CSP produced by $\mathtt{M}_0$ which models the $n$-Queens problem in $\mathcal{L}$ with $n \in \mathbb{N}$ fixed. The CSPs produced by $\mathtt{M}_0$ have a quadratic size in terms of $n$ (cf. the $3 \times \binom{n}{2}$ constraints) and use non linear constraints over integer variables.

As in other programing paradigms (functional programming, object oriented programming, etc.) one problem can be written as many (syntactically) different constraint programs with equivalent semantics (*i.e.*, they are all equivalently satisfiable or unsatisfiable or they all find the same optimal solution). We discuss this problematic in Section 2.3.2.

### 2.2.5 Modeller.

According to the type of variables (*e.g.*, Boolean, integer, continuous variables) and the type of the constraints (*e.g.*, linear, convex, non-linear, global constraints) one may look for the most appropriate research axes for modelling its problem. With an objective to share a common modelling language the mathematical programming community proposed A Mathematical Programming Language (AMPL for short) [10] as an algebraic modelling language for describing CSPs. AMPL is supported by dozens of state-of-the-art tools for constraint program solving (*e.g.*, CPLEX [11], Couenne [12], Gecode [13], JaCoP [14]). However, each research axe (each one specialized on dedicated families of CSPs) developed its proper modelling languages and tools. We present a landscape of CSPs families with their respective modelling languages and tools.

1. `SAT` (for Boolean Satisfiability Problem) contains CSPs with contraints over Boolean variables. The Conjunctive Normal Form (CNF for short) which consists of conjunctions of disjunctions of literals (*e.g.*, $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$ where $x_1$, $x_2$, and $x_3$ are three Boolean variables) is the main practical modelling language used in this community. The DIMACS [15] format is the standard text format for CNF representation. See [16] for more details about CNF encodings.

2. `LP`, `IP`, `MILP` (respectively for Linear Programming, Integer Programming, and Mixed-Integer Linear Programming) contain constraint programs with respectively: linear constraints over continuous variables; linear constraints over integer variables; and linear constraints over continuous and integer variables. These three families are identified as mathematical programming languages. Formally the constraint programs of these families are presented in the following form: $\mathbf{Ax} \leq \mathbf{b}$ where $\mathbf{x}$ is a column vector of variables with height $n$, and $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{b} \in \mathbb{R}^{m,1}$ are two matrices of coefficients. This encodes $m$ inequalities over $n$ variables. [17] recalls various modelling technics and use for these CSPs families.

3. `FD` (for Finite Domain Programming) contains constraint programs with constraints over variables with finite domains. There is no restriction on the constraints types. They can be linear, convex, non convex, non-linear such as trigonometric functions, exponential. There are also richer constraints expressed in the form of predicates, known as global (or meta) constraints which have been identified for their expressiveness (*e.g.*, `all-different`, `element`, `global-cardinality`) and help the solution process (cf. Section 2.3.1). See the Global Constraint Catalogue for more details [18]. Finally, there are two main formats for representing CSPs (XCSP3 [19] and FlatZinc [20]) each one associated with a constraint modeller (resp. MCSP3 [21] and MiniZinc [22]). While this CSP family allows any logical combination of constraints (negation, disjunction, implications, etc.) `FD` solvers are called propagation based solvers and are specialized for solving conjunction of constraints [23].

4. `SMT` (for Satisfiability Modulo Theory) allows any logical combinations of constraints over continuous and integer variables. The satisfiability stands for the logical combination of constraints while the theory stands for the semantics of the combined constraints. The logical combination of constraints can use any logical constraints (*i.e.*, conjunction, disjunction, negation, implication, equivalence). Theories range from linear-constraints to non-linear constraints with integer, real-number, Boolean, or any combination of these types (even bit vectors and floating-point numbers). The SMT-LIB format [24] is the standard format for representing CSP from this family. This norm also describes all the standard theories and their dependencies. `SMT` is more general than `SAT`, `IP`, `LP`, `MILP`. See [25] for an introduction to and applications of `SMT`.

**Example 5** (n-Queens Problem Continued)**.** We proposed in Example 4 the modelling $\mathtt{M_0}$ for solving the $n$-Queens problem. This modelling can be transformed into a linear integer modelling $\mathtt{M_1}$ producing CSPs from the `IP` family by replacing the non-linear constraints $|(\ell_i - \ell_j)/(c_i - c_j)| \neq 1$ by the constraints $\ell_i - \ell_j \neq c_i - c_j$ and $\ell_i - \ell_j \neq c_j - c_i$. Furthermore, this modelling can be transformed into a `FD` modelling $\mathtt{M_2}$ by replacing the $2 \times \binom{n}{2}$ constraints ensuring the "no threat" requirement by lines and columns with the only two following constraints: $\texttt{all-different}(\ell_1, \ldots, \ell_n)$ and $\texttt{all-different}(c_1, \ldots, c_n)$. Thus, $\mathtt{M_2}$ models are smaller than those from $\mathtt{M_0}$ and $\mathtt{M_1}$ Consider the 5-Queens problem. $\mathtt{M_0}$, $\mathtt{M_1}$, and $\mathtt{M_2}$ respectively produces 30, 40, and 22 constraints and have 10 variables. In addition to having less constraints, the models produced by $\mathtt{M_2}$ use the `all-different` global constraint which ensures faster resolution than the use of a clique of binary inequality constraints.

Example 5 illustrates how our $n$-Queens problem can be supported by the `IP` and the `FD` families. Thus, a constraint based problem can be modelled as many constraint programs such that each one can be targeted to possibly different constraint satisfaction program families. In the next section we explain how the different CSPs families are solved.

## 2.3 Constraint Solving

In this section we give an overview of CSP solving. While we presented in the previous section various CSP families for modelling constraint based problem, we present in this section how these families are solved in practice.

**Remark** We present in this section some general methods for solving the CSPs families presented previously. However, before using the general solution one may also check if its problem does not belong to a subfamilies with practical/theoretical advantages.

For instance, SAT community uses the Post's lattice for differentiating clones of Boolean functions for whose the satisfaction problem is in P or in NP [26]. In FD these complexity differentiation are dichotomy theorems, one famous is the Schaefer's dichotomy theorem [9]. Finally, in the non-linear programming context, the quadratic convex programming is in lower complexity class than non-linear programming [27].

### 2.3.1 Satisfaction

In the general case, the combinatorics implied by the relations between the variables in the constraints makes a CSP hard to solve. In practice, complete solvers need to explore the search space (*i.e.*, the set containing all the valuations). This is performed by branch and reduce algorithm where the search space is explored by developing a dynamic tree construction. Each node in the tree corresponds to a state in the exploration process (*i.e.*, a succession of choices/decisions leading to a partial valuation of the variables and/or a reduction of the domains size and/or the adjunction of learned knowledges, etc.). Thus, a path from the root to a leaf recursively cut the search space into smaller search spaces until the satisfiability or the unsatisfiability is proven [28, 29]. The search starts from the root node which consists of the original CSP to solve (*i.e.*, all the valuations are candidates possibly satisfying the CSP). Then, for each node in the tree exploration process the algorithm starts by reducing the current search space. This mainly consists in applying inferences rules such as resolution rules, computing consistency in order to reduce the search space while preserving all the valuations satisfying the CSP. Thanks to these reductions the next step checks if the reduced CSP is trivially satisfiable or unsatisfiable (*e.g.*, the CSP has been syntactically reduced to a tautology, a contradiction, an empty set of constraints, etc.). If the CSP is trivially satisfiable, then a valid valuation can be found (mainly by reading the domains which has been reduced thanks to the successive cuts). containing the decisions history. If the CSP is trivially unsatisfiable, then this exploration path is closed and the exploration process carries on in an other opened exploration path. Otherwise, the current state space is split into possibly 2, 3, ..., $n$ smaller search spaces and the exploration process will be evaluated for each smaller CSP instances.

Algorithm 1 recalls this generic search strategy. The two main generic functions are REDUCE and SPLITSEARCHSPACE which respectively reduces the search of the CSP while preserving all the valuations satisfying it (*i.e.*, this function may only remove unsatisfying valuations), and split the current CSP in many $k$ CSPs (with a possibly different $k \in \mathbb{N}$ at every loop iteration) such that the union of their search spaces cover the whole search space of the split CSP (it is not required to perform a partitioning and sub-problems may share valuations). Also, we considered here a queue as a CSP buffer but a more sophisticated object may be used to select dynamically the next buffered CSP to the treated. The algorithm stops when it finds a valuation satisfying one sub-problem. We now discuss how this generic is implemented for treating CSPs from various famillies.

```
1: function SATISFACTION(P = (X, D, C) : CSP) return Map<X,D>
2:     queue : Queue<CSP>
3:     P' : CSP
4:
5:     queue.add(P)
6:     while not(queue.empty()) do
7:         P' ← queue.pop()
8:         # Reduces the CSP while preserving all solutions
9:         P' ← REDUCE(P')
10:
11:        # Case the CSP is trivially satisfiable after reduction: returns a sat valuation
12:        if ISTRIVIALLYSAT(P') then
13:            return FINDVALUATION(P')
14:
15:        # Case the CSP is trivially unsatisfiable after reduction: ignores it
16:        else if ISTRIVIALLYUNSAT(P') then
17:            continue
18:
19:        # Else split the current CSP in "smaller" CSPs and add them to the queue
20:        else
21:            queue.addAll(SPLITSEARCHSPACE(P'))
22:
23:        end if
24:    end while
25:
26:    return ∅
27: end function
```

Algorithm 1: Generic Algorithm for Solving Constraint Satisfaction Problems

- In the SAT community the DPLL algorithm [30] corresponds to the instantiation of SPLITSEARCHSPACE by the selection of a non-instantiated variable $x$ (*i.e.*, a variable $x$ with domain $\{\mathsf{true}, \mathsf{false}\}$) and to the construction of two CSPs such that the first one contains the clause $x$ and the second one contains the clause $\neg x$. Then, the REDUCE function performs unit propagation and pure literal elimination. The ISTRIVIALLYSAT function checks if constraints form a consistent set of literals and ISTRIVIALLYUNSAT function checks the emptiness of the set of constraints.

- In the FD community, the constraint propagation with backtracking method consists in instantiating SPLITSEARCHSPACE and REDUCE in the following way. In general, SPLITSEARCHSPACE starts by selecting a non-instantiated variable $x$. Then, it constructs one CSP per value $k$ in the domain of $x$ such that each constructed CSP is derived from the current CSP by setting the domain of $x$ to the singleton domain $\{k\}$. We call search strategy a heuristic returning for a given CSP the next variables and domain values to use in order to realize the split search. On the other

hand, the REDUCE function performs propagations by calling filtering algorithms and computing consistencies (*e.g.*, node consistency, arc consistency, path consistency). Informally, a filtering algorithm removes values that do not appear in any solution. Global constraints usually come with dedicated filtering algorithms empowering the propagation process. Finally, the ISTRIVIALLYSAT function checks if the instantiated variables satisfy all the constraints and the ISTRIVIALLYUNSAT function checks if a constraint is violated or if a variable domain becomes empty. See [23] for more details.

- The branch-and-reduce framework used for solving non-linear programs with continuous variables, for instance HC4 [31], corresponds to instantiate in Algorithm 1 the function SPLITSEARCHSPACE by the branching function (*e.g.*, select a variable $x$ with domains $[a, b] \subseteq \mathbb{R}$ and a real number $c \in [a, b]$ in order to construct two CSPs which respectively contain the constraints $x \leq c$ and $x \geq c$), the REDUCE uses reducing consistencies in order to filter domain variables while preserving solutions. Finally, the ISTRIVIALLYSAT function guesses a valuation satisfying all the constraints for tight domains and the ISTRIVIALLYUNSAT function checks if a constraint is violated or if a variable domain became empty.

- The `SMT` community gathers solving techniques from various CSP families. Indeed, a `SMT` instance is considered as the generalization of a Boolean `SAT` instance in which various sets of variables are replaced by predicates (*e.g.*, linear or non-linear expressions for continuous variables, integer constraints). Thus, the SPLITSEARCHSPACE enumerates solutions of the `SAT` instance abstracting the CSP to solve (*i.e.*, each constraint which is not a Boolean function is replaced by a unique Boolean variable). Then, each solution of the `SAT` instance is translated into a set of constraints which leads to a conjunction of constraints, each one being a sub-problem to be solved. According to the *theory* of this sub-problem (linear programming, integer programming, etc.) proper methods from the corresponding CSP family are used. This approach, which is referred to as the *eager approach* loses the high-level semantics encoded in the predicates. Actual `SMT` solvers now use a *lazy approach* solving partial `SAT` sub-problem and then answering their corresponding predicate parts while constructing a global solution [32]. In `SMT`, we call *strategy* an implementation/construction of the SPLITSEARCHSPACE and REDUCE functions.

Furthermore each community (*i.e.*, `SAT`, `FD`, etc.) provides many tools which implement the generic Algorithm 1. The performance of these tools is mainly due to their implementation of the REDUCE and SPLITSEARCHSPACE functions inherited from years of research next to the practical resolution of real world problems. This includes the study and the availability of wide variety of concrete heuristics [33] and search strategies [34] eventually branched with an offline or online learning (*e.g.*, no good, learned clauses). In

this thesis we focus on constraint modelling of constraint based problems and on domain reducing functions called *propagators* implemented in the REDUCE function.

We presented independently how various CSPs families tackle the solving problem. However, some research has been realized in order to make them collaborate. For instance, the finite domains CSP family met continuous domains CSP family while preserving global constraints by linking CHOCO and IBEX solvers [35]. Also, the integration of both `IP` and `FD` has been discussed helping to design a system such as SIMPL [36]. On the other hand, we already mentioned the fact that the `SMT` community uses solving techniques for clearly identified theories. In the same time, they started to include global constraints from `FD` and [37] shows how the `all-different` constraint can be supported by `SMT` solvers which offers promising results. Finally, in [38] the authors develop cooperative constraint solver systems using a control-oriented coordination language. This work has been used for solving non-linear problems [39] and interval problems [40] as well.

We presented here a generic complete algorithm answering the constraint satisfaction problem. Such complete method always returns a valuation satisfying the given CSP if it exists and returns none if such valuation does not exist. Thus, an incomplete algorithm may not be able to indicate if the CSP is unsatisfiable but may find a valuation satisfying the constraint program. We consider complete solvers in this thesis.

### 2.3.2 Improving Models

As said in the modelling section there is more than one CSP which encodes a given constraint based problem. Furthermore, the time required for solving these equivalent CSPs may differ from one to another with possibly an exponential gap. We present here various methods exploring how CSPs can be improved for reducing solving time: reformulation, symmetry-breaking, redundant constraints, relaxation, and over-constraint. In all cases these improvements can be performed by hand. However, solvers may implement them for automatic uses.

**Definition 2.3.1** (Reformulation)**.** *Le $\mathcal{C}$ be a* CSP*. A reformulation $\rho$ transforms a* CSP *$\mathcal{C}$ into a* CSP *$\mathcal{C}'$ s.t. all the solutions of $\mathcal{C}$ can be mapped to a solution in $\mathcal{C}'$ and all the solutions of $\mathcal{C}'$ are translatable as a solution $\mathcal{C}$. Thus, $\mathcal{C}'$ models the same problem than $\mathcal{C}$.*

There exists model transformations for transposing a CSP from a family to another one. Definition 2.3.1 recalls the concepts of reformulation, *i.e.*, how to produce a new CSP from an existing CSP modelling the same contraint based problem. A low level constraint language such that the Conjunctive Normal Form (*i.e.*, a Boolean functions expressed as conjunctions of disjunctions of literals) language used in the `SAT` community is now tractable with `SAT` state-of-the-art solvers for millions of variables and constraints [41]. In some cases reformulating a satisfaction problem into a lower constraint language may offer better resolution times. For instance in [42], the authors reformulate their modelling

from CP to SAT which highly increases the sizes of the CSP. But the first modelling is not solved by CP solvers while the second one is solved by SAT solvers.

**Example 6** ($n$-Queens Refomulation)**.** Consider the modelling $\mathtt{M}_2$ presented in Example 5. Recall that the "no threat" rule for diagonals is managed by the constraints $\ell_i - \ell_j \neq c_i - c_j$ and $\ell_i - \ell_j \neq c_j - c_i$ considered for each pair of two different queen indexes $i$ and $j$. These constraints are equivalent to $\ell_i - c_i \neq \ell_j - c_j$ and $\ell_i + c_i \neq \ell_j + c_j$. Since they must hold for each pair of two different queen indexes, all these constraints can be replaced by the two `all-different` global constraints. Since the `all-different` constraint support variables as inputs we create the auxiliary variables $x_i$ and $y_i$ for all $i \in \{1, \ldots, n\}$ such that $x_i = \ell_i - c_i$ and $y_i = \ell_i + c_i$. Thus, the constraints expressing the "no threat" rule for diagonals can be replaced by the two constraints $\texttt{all-different}(x_1, \ldots, x_n)$ and $\texttt{all-different}(y_1, \ldots, y_n)$. We call $\mathtt{M}_3$ this modelling derived from $\mathtt{M}_2$. $\mathtt{M}_3$ is called a reformulation of $\mathtt{M}_2$. $\mathtt{M}_3$ models contain $n + 3$ constraints whereas $\mathtt{M}_2$ models contain a quadratic number of constraints in term of the number of queens $n$. Note that a reformulation may also change the variables and their domains and is not restricted to constraint modifications.

**Definition 2.3.2** (Symmetry)**.** *Le $\mathcal{P}$ be a constraint based problems over a set of objects $A$ with states $S$. We say that $\mathcal{P}$ contains symmetries iff there exists a permutation $\sigma$ of the set of configurations s.t. $\mathcal{P}$ is stable by $\sigma$. (*i.e.*, $\sigma(c) \in \mathcal{L}$, for all $c \in \mathcal{L} \subseteq S^A$ s.t. $\mathcal{L} \equiv \mathcal{P}$.)*

A symmetry in a constraint based problem is a permutation of the configurations in the problem (cf. Definition 2.3.2). Thus, symmetry breaking consists in taking advantages of symmetry detection in constraint based problem to only model a subset of all the configurations in the problem, *i.e.*, to only model the configurations which can not be obtained by symmetries. Symmetry breaking reduces the size of the search space and therefore, the time wasted in visiting valuations which are symmetric to the already visited valuations. The solution time of a combinatorial problem can be reduced by adding new constraints, referred as symmetry breaking constraints. We invite the reader to consider [43] for more details.

**Example 7** ($n$-Queens Symmetry Breaking)**.** Consider the modelling $\mathtt{M}_3$ presented in Example 6. Let $n$ be a fixed number of queens and $\mathcal{C}$ be the CSP produced by $\mathtt{M}_3$ for the $n$ queens problem. Note that in $\mathcal{C}$ the $n$ queens are unordered: *i.e.*, all the queens are totally identical. Thus, for any valuation solution of $\mathcal{C}$ one may interchange the values $(\ell_i, c_i)$ representing the position of the $i$th queen with the values $(\ell_j, c_j)$ representing the position of the $j$th queen to obtain another valuation solution of $\mathcal{C}$. We construct a new modelling, named $\mathtt{M}_4$, ordering the queens and realizing some symmetry breaking. $\mathtt{M}_4$ is such that for each $n \in \mathbb{N}$ its corresponding CSP model $(X, D, C)$ for solving the

$n$-Queens problem contains the $c_i$ variables with domain $\{1, \ldots, n\}$ and no variable $\ell_i$. Similarly to the previous modellings the $c_i$ variables represent the column position of the queens. However, in this modelling each $c_i$ with $i \in \{1, \ldots, n\}$ is fixed with a line, the $i$th line. Thus, $c_i$ contains the column position of the queen on the $i$th line and there are no more $\ell_i$ variables in $\mathtt{M_4}$. The constraints $x_i = \ell_i - c_i$ and $y_i = \ell_i + c_i$ from $\mathtt{M_3}$ are respectively replaced by $x_i = i - c_i$ and $y_i = i + c_i$ in $\mathtt{M_4}$ for all $i \in \{1, \ldots, n\}$. To sum up, the constraints in $C$ are the following ones: $\texttt{all-different}(c_1, \ldots, c_n)$, $\texttt{all-different}(x_1, \ldots, x_n)$, $\texttt{all-different}(y_1, \ldots, y_n)$, $x_i = i - c_i$ and $y_i = i + c_i$ for all $i \in \{1, \ldots, n\}$. Note that this modelling still contains symmetries (*e.g.*, chessboard rotations, chessboard plane symmetries). For instance setting the domain of the variable $c_1$ to $\{1, \ldots, \lceil n/2 \rceil\}$ removes some chessboard plane symmetries.

**Definition 2.3.3** (Redundancy)**.** *Let $\mathcal{C}$ be a* CSP *and $c$ be a constraint over a set of variables $X$. We say that $c$ is a redundant constraint for $\mathcal{C}$ iff adding the constraint $c$ to the* CSP *$\mathcal{C}$ does not change the solution space of $\mathcal{C}$.*

In a general context minimizing the number of constraints (and/or the number of variables) in a CSP does not necessary implies lower solving time in practice. We call redundant constraint a constraint which does not change the set of valuations satisfying a given CSP when adding this constraint to the CSP (cf. Definition 2.3.3). In practice adding well chosen redundant constraints may speed up the solving process, or obtain a scale up (see [44] for instance). However, in the linear programming case detecting redundant constraints in order to remove them may accelerate the resolution process (see [45] for more details).

**Definition 2.3.4** (Relax & Over-constrain)**.** *Let $\mathcal{P}$ be constraint based problem over a set of objects $A$ and a set of states $S$. Let $\mathcal{C}$ be a* CSP*. We say that $\mathcal{C}$ is:*

- *a* relaxed modelling *of problem $\mathcal{P}$ iff there exists a constraint based problem $\mathcal{P}'$ s.t.* CSP *$\mathcal{C}$ models problem $\mathcal{P}'$ and all the configurations in $\mathcal{P}$ belongs to $\mathcal{P}'$;*
- *an* over-constrained modelling *of problem $\mathcal{P}$ iff there exists a constraint problem $\mathcal{P}'$ s.t.* CSP *$\mathcal{C}$ models problem $\mathcal{P}'$ and all the configurations in $\mathcal{P}'$ belongs to $\mathcal{P}$.*

As last modelling strategies we present the relaxation and the over-constrain cases (cf. Definition 2.3.4). A relaxation is an over-approximation of a difficult problem by a nearby problem that is easier to solve. For instance a relaxation may transform integer variables into real variables (indeed this produces a greater solution space), or may only consider a convex hull of the problem by using only linear inequalities (*e.g.*, [46]). An over-constrain model is an under-approximation. This consists in modelling a constraint based problem contained by the original problem to solve. Thus, by reducing the size of the search space, one may hope a gain in term of resolution time (see [47] for instance).

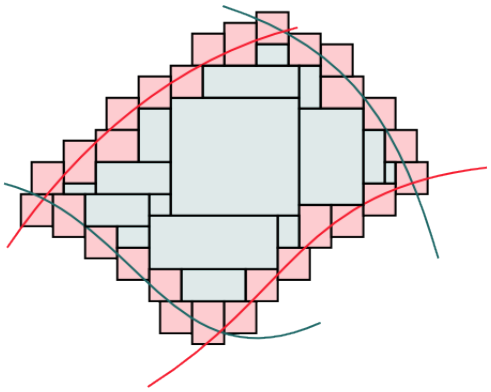


Figure 2.3: Box Paving for a Constraint Satisfaction Problem. Gray boxes only contain solutions. Pink boxes contains at least one solution. The union of the gray and the pink boxes covers all the solutions.

### 2.3.3 Real numbers vs. Floating-point numbers

As presented previously, one common variable domain for modelling is the real-numbers domain, written $\mathbb{R}$. When using this domain one expects that the solver takes into account the classical arithmetic properties verified by $\mathbb{R}$ (*e.g.*, associativity, commutativity, infinite limits, etc.). In practice, computer scientists use floating-point numbers for simulating real numbers. Floating-point numbers represent a finite numbers of real numbers with finite (binary) representation. The IEEE 754 norm [48] is now considered as the norm for representing floating-point numbers in programs. This norm encodes $2^{32}$ finite real numbers where the smallest non-zero positive number that can be represented is $1 \times 10^{-101}$ and the largest is $9.999999 \times 10^{96}$, the full range of numbers is $-9.999999 \times 10^{96}$ through $9.999999 \times 10^{96}$; it contains two signed zeros $+0$ and $-0$, two infinities $+\infty$ and $-\infty$, and two kinds of $NaN$s. In the following we write $\mathbb{F}$ for the set of real-numbers representable in the IEEE 754 norm.[3] The first notable fact is that floating-point arithmetic (*i.e.*, arithmetic over $\mathbb{F}$) is not equivalent to real number arithmetic (*i.e.*, arithmetic over $\mathbb{R}$). Precision limitation with floating point numbers implies rounding: a real-number $x \in \mathbb{R}$ which is not in $\mathbb{F}$ is rounded to one of the neartest floating-point number. The IEEE 754 norm describes five rounding rules (two rules round to a nearest value while the others are called directed roundings and round to a nearest value in a direction such as 0, $+\infty$, $-\infty$). For instance: $0.1_{10}$ (number 0.1 represented in base 10) does not have a finite representation in base 2 and thus, it does not belong to $\mathbb{F}$; there exists floating-point numbers $x \in \mathbb{F}$ s.t. $x + 1 = x$ in the floating-point arithmetic.

Implementing real number arithmetic in CSP solvers is challenging. Recall that we presented CSP valuation solutions as a mapping from the variables to their respective domains. Since some valuations to $\mathbb{R}$ may not be representable with floating-point numbers, solvers like RealPaver [49] find reliable characterizations with boxes (Cartesian product of

[3] We chose the IEEE 754 norm but any set of floating-point numbers can be considered.

intervals) of sets implicitly defined by constraints such that intervals with floating point bounds contain real-number solutions. Thus, the real number valuation solutions are bounded by the interval valuation solutions.

Chapter 3

# Program Verification

## Contents

Since softwares take more and more control over complex systems with possibly critical impact on the society (*e.g.*, car driving software, automatic action placements, ...) the verification community develops methods for ensuring the validity of such programs. After introducing in a first section the main objectives of program verification, we go deeper into the two verification fields concerned by our contributions. Our first contribution relates to Abstract Interpretation and the second one considers Model Checking for Markov chains. Finally, we present a brief overview concerning how constraint programming meets verification problems.

**Warning.** We choose the word "program" for system change descriptions while the verification community also uses the word "model" with the same signification. Recall that we already introduced the word "model" in the constraint programming background (cf. Section 2.2). Thus we reserve it for the constraint context.

## 3.1 Introduction

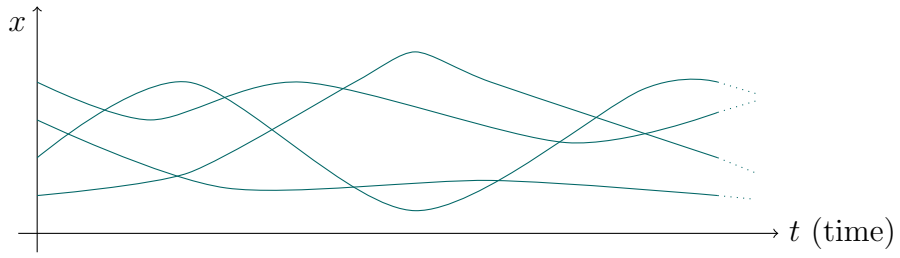



Figure 3.1: Instance of four possible traces of a variable $x$ while executing the same program.

We focus in this thesis on program verification problems, *i.e.*, we do not consider hardware verification problems. In this context, the word "program" refers to a computer science program written in a dedicated programming language [50]. There is a wide variety of programming languages which can be grouped by programming paradigms: functional programming (*e.g.*, Javascript, Python), object oriented programming (*e.g.*, C++, Java), reactive programming (*e.g.*, FAUST, LUSTRE), probabilistic programming (*e.g.*, ProbLog, RML), etc. Even if these languages may have different programming approaches they all share the same verification expectations. Indeed, whatever the language, a program is designed to be executed (in our concern we consider that programs are executed on a machine with memory). We briefly recall some vocabulary proper to program verification. We call *run* an execution of a program. During a run the machine *memory* varies over the time. We call *state* a snapshot of the memory at a given time. Finally, a *trace* is the succession of states corresponding to a run of the program. Thus, program verification consists in analyzing traces in order to determine if a given *property*

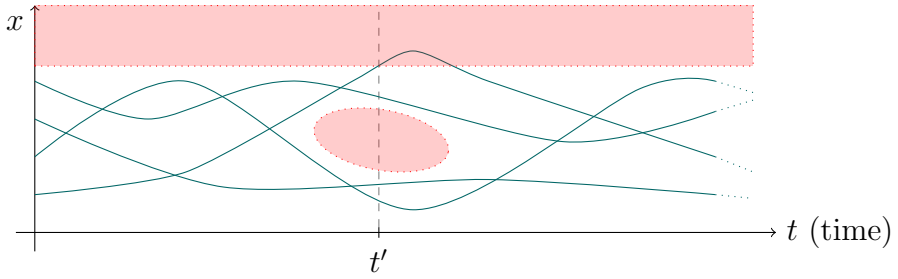



Figure 3.2: Instance of four possible traces of a variable $x$ while executing the same program.

is satisfied. We name *concrete semantics* the set of all the traces of a given program. The variations for a single program between its traces may come from user inputs given at running time, non determinism of the program, probabilistic transitions in the program, etc.

**Example 8.** Figure 3.1 describes a simple program using one variable $x$ as memory. Here, the concrete semantics of this program contains exactly four traces. Each trace is represented as a curve on the graphics such that each time step $t$ corresponds to the value of $x$ at this time.

Verification consists in verifying properties on the concrete semantics of programs. Recall first that these semantics are an "infinite" mathematical object (*i.e.*, an infinite set of potentially infinite sequences of states) which is not computable: it is not possible to write a program able to represent and to compute all the possible traces of any program. Otherwise, one may also solve the halting problem [51]. Thus, in the general case, questions about the concrete semantics of a program are *undecidable*. In practice the program traces may be *finite*. However note that in this thesis both contributions only consider *infinite traces*. Properties on such traces may be expressed using different formalisms. First, they can be time independent. In such case the property must hold during each execution time (*e.g.*, the variable $x$ must never be equal to zero, the variable $x$ must always take its values between $-1$ and 1): such properties are called invariants. Secondly, time dependent properties express how the memory must vary over time for each trace (*e.g.*, the variable $x$ must not be equal to zero before a given line of the program, the variable $x$ must be bounded by $-1$ and 1 at the end of the execution, using temporal modalities such as the linear temporal logic [52]). Finally, a verification process has the objective to determine according to a given program and a property if this program satisfies this property. Recall that the program verification problem is undecidable in the general case (the halting problem can be turned into a verification problem). Thus, such verification process may *validate*, *invalidate*, or be *non conclusive* concerning the satisfaction of the property by the program.

We now present two main approaches, named static analysis and dynamic analysis, for tackling verification problems. Roughly speaking, the first one may be seen as offline verification and the second one as online verification. We call *static analysis* a verification process working without explicitly running the programs. On the other hand, a *dynamic analysis* verification process requires to run the program in order to validate or invalidate a property. Furthermore, both approaches also consider bounded verification vs. unbounded verification. Bounded verification only checks the validity of the properties for sub traces (*i.e.*, for a bounded time range of execution of the program) while unbounded verification checks the validity of the properties for all traces regardless of their length. Recall that the total set of traces of a given program is called the concrete semantics of this program. Given a concrete semantics an abstraction is a mathematical model (possibly a program) representing at least all the traces in the concrete semantics. Thus, the verification process may be performed with an abstraction of the program instead of the original program itself. We say that a verification process is *sound* w.r.t. to a program abstraction iff it agrees with the verification of the concrete domain. Otherwise this process is called *unsound*.

**Example 9** (Example 8 continued)**.** Figure 3.2 contains the same program traces as presented in Example 8. The red areas represent "errors" and are also called *unsafe regions*. Note that the top red region is time independent (for all the running time, the variable $x$ must not be greater or equal to a certain value). Conversely, the other red region is time dependent. We want to verify that no program trace enters in an unsafe region. This property is not valid if at least one trace in the concrete domain reaches an unsafe region. Here, since one trace overlaps one red region, this program does not satisfy this property. Thus, a sound static analysis will validate the property for this program. Note that a dynamic analysis bounding the verification process with a limited horizon could pass next to the trace violating the property and miss this counter example. As a consequence, bounded verification processes might be unsound.

## 3.2 Abstract Interpretation

Abstract Interpretation [53, 54] (AI[1] for short) does static program analysis by building abstract representations of the program behaviors. The first contribution of this thesis (Chapter 4) is inspired from AI. We present in this section the general ideas behind AI and we conclude by citing related works linking AI and CP. Example 10 motivates the main ideas behind Abstract Interpretation by the use of an example: bounding the variables at every step in the program.

[1]Be careful that in this thesis we use AI for Abstract Interpretation and not Artificial Intelligence.

```
1: int x, y
2: y ← 1
3: x ← random(1, 5)
4: while y < 3 and x < 5
   do
5:    x ← x + y
6:    y ← 2 * y
7: end while
8: x ← x + 1
```

(a) Running Program Example

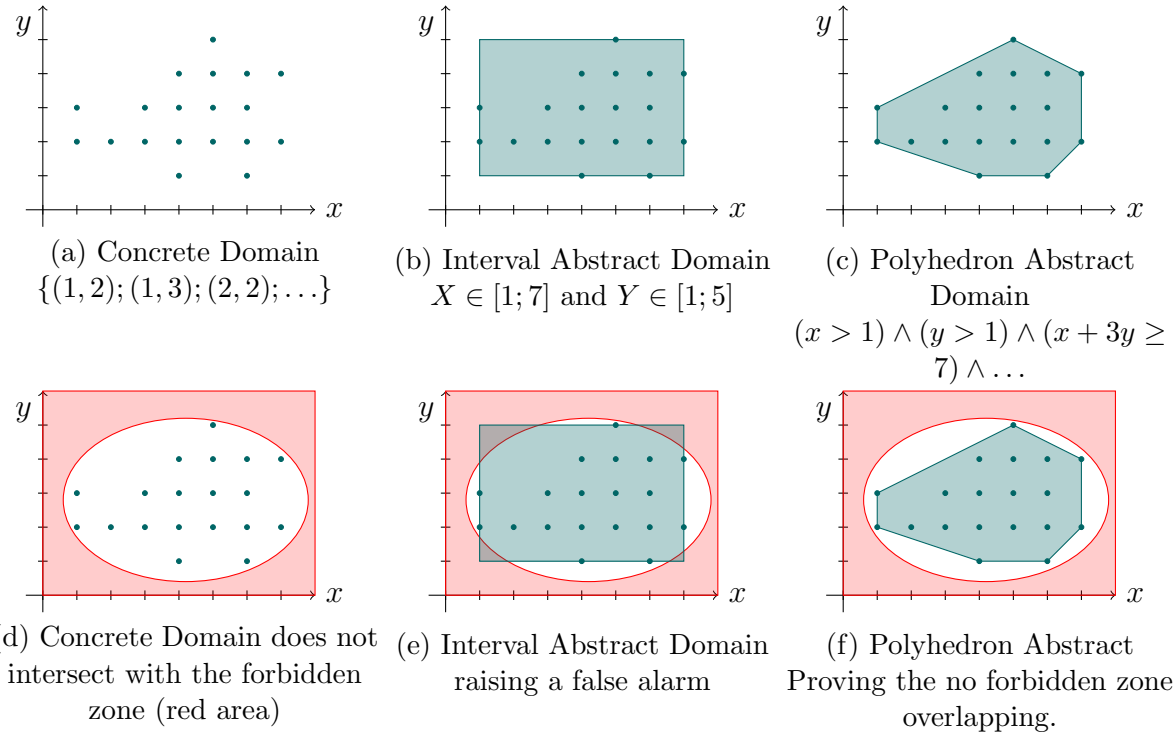
| Trace | line 3 | line 5 | line 6 | line 5 | line 6 | line 8 |
|---|---|---|---|---|---|---|
| $trace_1$: | $(1,1)$ | $(2,1)$ | $(2,2)$ | $(4,2)$ | $(4,4)$ | $(5,4)$ |
| $trace_2$: | $(2,1)$ | $(3,1)$ | $(3,2)$ | $(5,2)$ | $(5,4)$ | $(6,4)$ |
| $trace_3$: | $(3,1)$ | $(4,1)$ | $(4,2)$ | $(6,2)$ | $(6,4)$ | $(7,4)$ |
| $trace_4$: | $(4,1)$ | $(5,1)$ | $(5,2)$ | | | $(6,2)$ |
| $trace_5$: | $(5,1)$ | | | | | $(6,1)$ |

(b) Program traces for the running program example where each pair contains the values for $x$ and $y$

Figure 3.3: Running program example with its five traces.

**Example 10.** Figure 3.3a contains a program described in pseudo code using two variables $x$ and $y$. Variable $y$ is initialized to 1 and variable $x$ is initialized to an integer randomly selected between 1 and 5. Then, the loop body is evaluated while the condition $y < 3 \wedge x < 5$ is true, and finally, the variable $x$ is incremented by 1. Clearly, this simple program admits 5 traces presented in Figure 3.3b. These different traces come from the random function on line 3 allowing 5 possible outputs. Note that traces may share states/state sequences (*e.g.*, state $(4,2)$ is in traces $trace_1$ and $trace_3$), traces may have different lengths (*e.g.*, $|trace_1| = |trace_2| = |trace_3| = 6$ while $|trace_4| = 4$). Also the number of traces may be exponential or even infinite. Recall that computing this number is an undecidable problem in the general case.

As said previously, a verification process is performed on an abstraction of the concrete semantic. In order to bound all the variables at every program step, AI uses a connexion between the concrete semantics and the so called *abstract semantics*. Regarding a program concrete semantic and a variable in this program, we call *concrete domain* the values taken by the variables at each program step. On the other hand, for the same program and variable an *abstract domain* contains a super set of the the variable concrete domain at each program step (*i.e.*, an abstract domain is an over-approximation of a concrete domain). While the concrete domain is unique there exist many possible abstract domains. Thus, AI proposes a variety of (mathematical) abstractions such that each one has advantage and disadvantage in term of precision, representativity, computability, etc. The problem of determining tight abstract domains becomes harder when the program contains loops. Indeed, this problem can be reduced to the search of inductive invariant (ideally the smallest). AI uses widening and narrowing operators in order to approach such solution. This is performed by considering an abstraction of the loop transfer function. Then, using this transfer function for testing over-approximation candidates and following successive widening and narrowing iterations terminates and converges to an inductive invariant. We invite the reader to consider [53, 54] for a formal and exhaustive presentation of AI since no more background is required for our contributions. Example 11




(a) Concrete Domain $\{(1,2);(1,3);(2,2);\ldots\}$

(b) Interval Abstract Domain $X \in [1;7]$ and $Y \in [1;5]$

(c) Polyhedron Abstract Domain $(x > 1) \wedge (y > 1) \wedge (x + 3y \geq 7) \wedge \ldots$

(d) Concrete Domain does not intersect with the forbidden zone (red area)

(e) Interval Abstract Domain raising a false alarm

(f) Polyhedron Abstract Proving the no forbidden zone overlapping.

Figure 3.4: A concrete domain (a) over variables $x$ and $y$ abstracted by an interval abstract domain (b) and a polyhedron abstract domain (c) respectively without (a,b,c) and with (d,e,f) a forbidden zone s.t. interval abstraction produces a false alarm (e) while polyhedron abstraction proves safety (f).

describes the link between concrete and abstract semantics and presents the interest of tight abstract domains to possible false positive. This example presents the interval and the polyhedron abstractions. There exists other abstract domains such as the sign and the octogone abstract domains. Furthermore, a precision gain with an abstract domain increases the theoretical complexity for maintaining it while progressing in the verification process. However the use of abstractions may offer a guarantee of termination of the verification process. Indeed, well chosen abstractions produce semi-decidable problems. Thus verification tools based on abstract interpretation either proves the satisfaction of the bounding property or cannot conclude.

**Example 11.** Assume that a program using two variables $x$ and $y$ admits at a program step the concrete semantics presented in Figure 3.4a. Thus, each dot in the figure corresponds to a state encountered by one or more program traces at the given program step. One possible abstraction for this concrete domain is the use of intervals to create a box (*i.e.*, a cartesian product of intervals). Figure 3.4b presents the smallest interval domain containing the concrete semantics. On the other hand, Figure 3.4c presents the smallest polyhedron domain containing the concrete domain. According to this concrete domain and these abstract domains consider the unsafe region as the red region presented in Figure 3.4d. Since none of the states in the concrete domain overlap the unsafe region, the

program satisfies the property. However, note that the interval domain intersects with the unsafe region in Figure 3.4e. Thus, this abstract domain is not able to prove the validity of the property on the concrete semantics. Conversely, the polyhedron abstract domain presented in Figure 3.4f successfully proves the validity of the property on the concrete semantics.

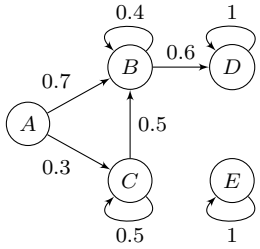
## 3.3 Model Checking

Model checking provides formal verification of properties over models/programs with finite structure that potentially abstract systems with infinite state spaces. In our contribution relative to model checking (Chapter 5) we consider programs which are representable as finite-state machines (more precisely Chapter 5 considers Discrete Time Markov chains). Informally, a finite-state machine contains a finite set of states together with a set of initial states and a set of final states. States are linked by a transition function determining possible state successions. Thus, a run is a (possibly infinite) succession of states from an initial state and its trace corresponds to this sequence of states.[2] Example 12 presents an example of Markov chain.

**Example 12.** Figure 3.5a contains an example of Markov chain containing the states $A$, $B$, $C$, $D$, and $E$, $A$ being the initial state. The edge labelling gives the probability to move from a state to another. Note that all the outgoing transitions from a given state form a probability distribution (*i.e.*, they are all positive real-numbers and they sum to one). Also, in our setting Markov chains do not have final states. Thus, we consider that the accepted runs are all the infinite sequences of states with non zero transition probabilities. Indeed, each run is associated with a probability corresponding to the product of all the probabilities encountered on the transitions. Figure 3.5b present the prefixes with size 4 of five infinite runs with their corresponding probability. The probability for the runs not starting from the initial state or including a missing transition is set to zero.

As presented in section 3.1 we consider system/program verification based on trace properties. Model checking (also named property checking) consists in exhaustively and automatically checking whether the model of a system meets a given specification/property. In the context of finite-state machines, such properties are expressed in order to discriminate infinite runs (*e.g.*, all the runs meeting state $B$ before state $C$, all the runs reaching $D$ before 5 transitions). Since the number of states is finite, and runs are infinite, the number of different runs is infinite but countable. The Linear Temporal Logic [52] (LTL for short) uses temporal operators (*e.g.*, *next*, *until*) allowing to define such sets of traces. Finally, *qualitative verification* and *quantitative verification* take into account a measure over the traces and consist in checking that the set of runs accepted by

[2] In the case of the presence of state labelling, the trace is the succession of labels associated to the states encountered at running time (cf. Chapter 5).




(a) Example of Markov chain

| Trace | #1 | #2 | #3 | #4 | Probability |
|---|---|---|---|---|---|
| $prefix_1$: | $A$ | $B$ | $B$ | $D$ | 0.168 |
| $prefix_2$: | $A$ | $C$ | $B$ | $D$ | 0.09 |
| $prefix_3$: | $A$ | $C$ | $C$ | $B$ | 0.075 |
| $prefix_4$: | $A$ | $C$ | $E$ | $E$ | 0 |
| $prefix_5$: | $E$ | $E$ | $E$ | $E$ | 0 |

(b) 5 trace prefixes with their respective probability to occur.

Figure 3.5: A Markov chain next to 5 trace prefixes with size 4, associated with their respective probability to occur.

a the state machine verifies the property quantified by the given measure. Example 13 illustrates qualitative and quantitative properties verification.

**Example 13** (Example 12 continued)**.** Consider the property asserting that the runs must encounter state $D$. This property does not hold for the Markov chain presented in Figure 3.5a. Indeed, there exist runs infinitely looping in state $B$ or $C$ which never encounter state $D$ (more precisely there exists an infinite number of such runs). On the other hand, consider the property asserting that the probability of encountering state $D$ equals 1: this property holds. Indeed, the probability of looping infinitely in state $B$ or $C$ equals to zero. Thus, all the runs with a non zero probability reach $D$ and the probability of reaching $D$ equals to 1.

Recall, that runs are infinite and the set of state is finite. Thus, model checking explores all possible system states in a brute-force manner. This way, it can be shown that a given system model formally satisfies or falsifies a certain property. Hence, such method proves the validity of the property or returns a counter-example otherwise.

## 3.4 Constraints meet Verification

Considerable improvements in the efficiency and expressive power of constraint program solvers allowed to tackle problems more and more difficult to answer. In this section, after motivating the use of constraint programming for answering the two mains program verification problems considered in this thesis we present various verification processes using constraint programming.

Even if a program admits a priori an infinite state space its executions may encounter a (potentially infinite) subset of the declared state space. Thus, one would like to determine this smaller state space in order to verify the non reachability of undesired states. This problem is reducible to the search of program over-approximations, *i.e.*, bounding all the program variables. This is an objective of Abstract Interpretation where the

program describing precisely the system evolution from a state to another, named the concrete program, is abstracted. This abstracted construction is related to the concrete program in such a manner that if an over-approximation holds for the abstraction then, this approximation also holds for the concrete program. Furthermore, constraint programs allow to describe over-approximations such as convex polyhedrons using linear constraints, ellipsoids using quadratic constraints, etc. Thus, since constraint programming is a generic declarative programing paradigm it may be seen as a verification process for over-approximating variable in declarative programs. In the first contribution, we consider a block-diagram language where executions are infinite streams and the objective is to bound the stream values using constraint programming.

However, bounding the state space is not enough for some verification requirements. In our second problem, the objective is to determine if a specific state is reachable at execution time. Indeed, abstractions can only determine if a specific state is unreachable. For this verification problem, we consider programs representable as finite graph structures where the nodes form the state space and the edges give state to state transitions. Thus, verifying the reachability of a state in such a structure is performed by activating or deactivating transitions in order to reach the target state. However, these activations can be restricted by guards, or other structural dependent rules. Clearly, this corresponds to a combinatoric problem to solve. For this reason, since one of the objectives of constraint programming is to solve highly combinatorial problems, the verification community is interested in the `CP` tools.

**Dynamic Analysis.** Software testing consists in checking the validity of a property on a given program by running simulations. The classical book *The Art of Software Testing* [55] defines software testing as "the process of executing a program with the intent of finding errors" (*i.e.*, finding runs which do not satisfy the specification). Thus, Constraint-Based Testing is the process of generating program test cases by using the constraint programming technology [56]. The test cases are not written by hand but constraint programming solvers are used to produce them. A recent survey for this research field can be found in [57].

**Static Analysis.** Static program analysis is the automatic determination of runtime properties of programs. This consists in finding run-time errors at compilation time without code instrumentation or user interaction. K. R. Apt formalized the link between *chaotic iterations* such as used in abstract interpretation for moving between fixed points or inductive invariants and the resolution process used in contraint programming [58, 59]. More recently, in [60], the authors integrate abstract domains into a constraint programming solver by developing a tool named Absolute.

**Bounded Model Checking.** Binary Decision Diagrams (BDDs) have been used for formal verification of finite state systems with success since their introduction in the beginning of the 90's. However, in [61] the authors proposed CNF modellings instead of BDD modellings for realizing Bounded Model Checking (BMC for short). This CNF based verification process takes advantages of the efficiency of the SAT solvers which are now considered as the state-of-the-art techniques for bounded model checking. This formulation in the BMC context led to a scale up in the size of the verified programs and also replaced the dedicated methods developed with BDDs.

**CP solvers for testing applications** More generally, constraint programming solvers have been used to test applications thanks to the expressiveness and the efficiency of constraint programming languages. For instance, [62] uses constraint propagation to check a cryptanalysis problem: by providing a better solution, they proved that a solution claimed to be optimal in two cryptanalysis papers was not optimal.

**Constraints in formal verification** More and more the verification community uses `SMT`s tools instead of dedicated algorithms for performing verification process. Indeed, the actual best state-of-the-art `SMT` solvers are able to handle linear, non-linear, and even quantified constrained programs which may appear in program verification problems. For instance, see [63] for symbolic software model checking or the Extended Static Checker (ESC) tool using the Simplify SMT solver [64].

This concludes our overview of constraint programming and program verification. After having put in perspective both research fields we now propose two chapters, each one self contained, about solving some program verification problems using constraint modelling and solving.

Chapter 4

# Verifying a Real-Time Language with Constraints

## Contents

This chapter treats the verification of synchronous languages modelled as block-diagrams. The verification problem consists in bounding all the variables in the program. This problem is called the stream over-approximation problem. We present a global constraint in the spirit of Constraint Programming designed to deal with this problem. We propose filtering algorithms inspired from abstract interpretation and prove their validity. These algorithms are inspired from both continuous constraint programming and abstract interpretation. Finally, we propose an implementation of our modellings and discuss the results. This chapter is self-contained including introduction, motivation, background, state of the art, and contributions.

## 4.1 Introduction

Constraint programing (CP) [65] offers a set of efficient methods for modelling and solving combinatorial problems. One of its key ingredients is the propagation mechanism, which reduces the search space by over-approximating the solution set. For continuous constraints [66, 67], propagation is defined in a generic way on a given constraint language, usually containing equalities, inequalities, and many operators (arithmetic operations, mathematical functions, etc). In this chapter, we present a method using this generic propagation scheme, combined with a new solving algorithm, for the resolution of a verification problem.

Our problem consists in checking the range of the outputs of programs written as block-diagrams, a common model for many real-time languages. More precisely, we are interested in DSP (Digital Signal Process) programs, based on a block-diagram algebra, which contains both typical real-time operations (split, merge, delay, ...) and mathematical functions [68]. All the variables are infinite streams over the reals. A stream represents the values taken by a variable at each time step. All the variables/streams are synchronized and they all receive a value at each tick of the clock. All the loops are thus, in theory, infinite by construction: the programs do never stop by themselves. Of course, they may stop computing in practice when all the signals are constant, or alternatively they can be killed by the user. The problem we tackle is the following: considering a block-diagram, which comes from a real-time program on streams, can we compute or approximate at a good precision the range of the stream output by this program?

This problem is, in a more general form, at the core of another research area, Abstract Interpretation, as introduced in [54, 53]. Abstract Interpretation offers a great variety of tools to over-approximate traces of programs to prove the absence of some runtime errors, such as overflows. It relies on abstractions of the program traces, *i.e.* the possible values the variables may take during an execution. The set of all the possible program traces cannot be computed in the general case. In Abstract Interpretation, they are represented by an abstract element, easier to compute, which must both include all the program

traces and be reasonably easy to compute. One of the first examples of such abstraction is the interval abstract domain [53], which is used in this work. An abstraction comes with several operators to mimic the program execution. Abstract Interpretation has been successfully applied to a wide range of applications the most famous one being the analysis of the flight-control commands of the Airbus A380 aircraft.

In this work, we use tools from constraint programming to compute precise abstractions of all the stream variables of a real-time program. Our method is generic and can be applied to any language based on a block-diagram algebra for bounding the values taken by the input, output, or inner streams of the program. We present three applications of our method: compiler assistance, refactoring, and verification. We chose the verification application for the experiment section that we applied on the FAUST (Functional Audio Stream)[1] language. This language has been designed for sound design and analysis [69] of audio streams. FAUST is a functional language with a proper semantic based on block-diagrams, which makes it a language quite similar to LUSTRE [70] or its commercial version SCADE. In practice, the compiler automatically generates a block-diagram for each program. The outputs of these block-diagrams are real-valued streams which represent audio signals. These signals can for instance be sent to loudspeakers or other applications.

By convention, digital audio signals must stay in the range $[-1, 1]$. In case the signal takes values out of this range, it can either damage the loudspeakers, or, more currently, be arbitrarily cut by the sound driver. In this case, the shape of the sound is modified and this produces a very audible sound effect called saturation. For this purpose, we compute bounds for the values taken by all the signals in the program and then we verify that the output streams stay in $[-1; 1]$. Verifying FAUST programs is essential since this language is intended for non computer scientists. An overflowing program not only produces a corrupted sound, but in practice, it often has conception mistakes. Moreover, FAUST programs are now used in concerts or commercial applications [71].

FAUST already embarks a static analyzer based on Abstract Interpretation, using the interval abstract domain. The analyzer computes the outputs of each operator from its inputs, with the bottom-up top-down (or HC4) algorithm. When the programs do not have loops or delays, this works very well. However, as soon as the programs have loops, the interval analysis cannot provide precise over-approximation (returning $[-\infty, +\infty]$) and the analysis fails.

In this chapter, we first propose a model of the verification of a block-diagram as a constraint problem. Propagation based on the constraints allows us to compute an over-approximation of the range of the computed streams. But as soon as the program has loops, the approximations are too large. We present a specific solving method which identifies the loops in the constraint graph, and propagates these constraints in a specific

[1] FAUST is open source and available at `http://faust.grame.fr`

way to find over-approximation with a better precision. We implemented this method on FAUST block-diagrams, using IBEX [72, 67] a constraint programming solver over continuous domains. We tested it on several programs from the FAUST library. Most of the times, the over-approximation returned by our method is optimal, in the sense that it is the best interval approximation. We have tested our method on the programs given as examples in the standard FAUST library, with good results: we were able to detect errors in two of these programs, and in general to fastly compute precise intervals over-approximating the program outputs.

This chapter is organized as follows: Section 4.2 introduces the notion of block-diagrams. Section 4.3 presents the conversion of block-diagrams into a first constraint model. Section 4.4 defines the global constraint used for a more efficient model. Different applications of our optimized model are presented in Section 4.5 and we consider one of them in Section 4.6 with our application language and present the results of the experimentation followed by related works. Finally, Section 4.7 discusses the contribution and future works.

**Related Works** The research on Constraint Programming and Verification has always been rich, and gained a great interest in the past decade. Constraint Programming has been applied to verification for test generation (see [57] for an overview), constraint-based model-checking [73], control-flow graph analysis [74] or even worst-execution time estimations [75]. More recently, detailed approaches have been presented by [76] or [77] to carefully analyze floating-points conditions with continuous constraint methods.

Other approaches mix CP and Abstract Interpretation. It has been known for a long time that both domains shared a lot of ideas, since for instance [59] which expresses the constraint consistency as chaotic iterations. A key remark is the following: Abstract Interpretation is about over-approximating the traces of a program, and Constraint Programming uses propagation to over-approximate a solution set. It is worth mentioning that one of the over-approximation algorithms used in Abstract Interpretation, the bottom-up top-down algorithm for the interval abstraction [54, 78], is the same as the HC4 constraint propagator [66] (in the following, we will refer to this algorithm as HC4), which shows how close CP and Abstract Interpretation can sometimes be. More recent works explored these links in both ways, either to refine CP techniques [79, 60] or to improve the Abstract Interpretation analysis [80, 81]. Finally GATeL [82] uses Constraint Logic Programming for verifying real-time programs by test cases generation.

In some sense, our work can be seen as solving a constraint problem on streams. There have been other works on stream constraints in the literature (*e.g.*, [83, 84]). However, this approach radically differs from ours because their stream constraints are meant to build an automaton whose paths are solutions of the constraints. In particular, we would not be able to analyze infinite streams in a non-regular language with these

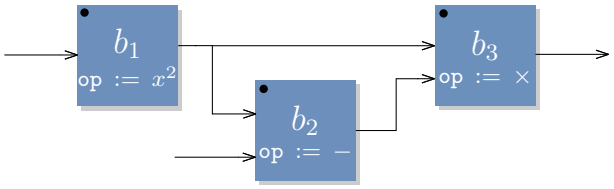



Figure 4.1: A block-diagram in $\mathtt{BD}(\mathbb{R})$

stream constraints. On the contrary, our constraints are expressed on infinite streams, and generated in order to compute hulls of the streams.

## 4.2 Background

This section introduces the block-diagram algebra for representing real-time programs.

### 4.2.1 Syntax

A block is a function that applies an operator on some ordered inputs, and generates one or more ordered outputs.

**Definition 4.2.1** (Block)**.** *Let $E$ be a nonempty set. A* block *over $E$ is a triple $b = (\mathtt{op}, n, m)$ such that: $n \in \mathbb{N}$ is the* number of inputs *of the block, $m \in \mathbb{N}$ is the* number of outputs*, and $\mathtt{op} : E^n \to E^m$ is the* operator *of the block. The $n$* inputs *and the $m$* outputs *are ordered: $[i]b$ refers to the $i$th input ($1 \leq i \leq n$) and $b[j]$ to the $j$th output ($1 \leq j \leq m$).*

For any block, we say that input $i$ (respectively output $j$) exists iff $i$ (resp. $j$) is an integer between 1 and the number of inputs (resp. outputs) of the block. Throughout this chapter, given a nonempty set $E$, $\mathtt{Block}(E)$ denotes the set of all the blocks over $E$.

**Definition 4.2.2** (Connector)**.** *Let $B$ be a set of blocks. A* connector *over $B$ is a pair $(b[i], [j]b')$ such that: $b$ and $b'$ are blocks from $B$; output $i$ exists for block $b$ and input $j$ exists for block $b'$.*

**Definition 4.2.3** (Block-Diagram)**.** *Let $E$ be a nonempty set. A* block-diagram *over $E$ is a pair $d = (B, C)$ such that: $B$ is a set of blocks over $E$ and $C$ is a set of connectors over $B$. An input (respectively output) of a block in $B$ that does not appear in a connector of $C$ is an input (respectively an output) of the block-diagram $d$.*

Similarly to the blocks, if a block-diagram $d$ has $n$ inputs and $m$ outputs, we can order them and: $[i]d$ refers to the $i$th input ($1 \leq i \leq n$), and $d[j]$ to the $j$th output ($1 \leq j \leq m$). Finally, we denote $\mathtt{BD}(E)$ the set of all the block-diagrams over $E$.

**Example 14.** Figure 4.1 depicts a block-diagram over real numbers in $\mathtt{Block}(\mathbb{R})$ containing three blocks: block $b_1$ has the square function as operator; block $b_2$ has the

subtraction, and block $b_3$ has the multiplication. Connectors are represented by arrows: connector $(b_1[1], [1]b_2)$ from block $b_1$ to block $b_2$; $(b_1[1], [1]b_3)$ from $b_1$ to $b_3$ and $(b_2[1], [2]b_3)$ from $b_2$ to $b_3$. This block-diagram has two inputs (*i.e.*, $[1]b_1$ and $[2]b_2$) and one output (*i.e.*, $b_3[1]$).

### 4.2.2 Semantics

After the syntax, it is natural to define the block-diagram semantics: block-diagram interpretation, and block-diagram model.

**Definition 4.2.4** (Interpretation)**.** *Let $E$ be a nonempty set, $b = (\mathtt{op}, n, m)$ a block in $\mathtt{Block}(E)$, and $d = (B, C)$ a block-diagram in $\mathtt{BD}(E)$. An* interpretation *$I$ of block $b$ is a mapping from each input $i$ to an element in $E$ (noted $I([i]b)$), and a mapping from each output $j$ to an element in $E$ (noted $I(b[j])$). An* interpretation *$I$ of the block-diagram $d$ is an interpretation of each block in $B$.*

An interpretation is any valuation of all the inputs and all the outputs. We introduce the notion of *model* to highlight the interpretations considering the operators (*i.e.*, such that the outputs correspond to the image of the inputs by the operators) and the connectors.

**Definition 4.2.5** (Model)**.** *Let $E$ be a nonempty set, $b = (\mathtt{op}, n, m)$ a block in $\mathtt{Block}(E)$, and $d = (B, C)$ a block-diagram in $\mathtt{BD}(E)$.*

- *An interpretation $I$ of block $b$ is a* model *of $b$ iff*
$$\mathtt{op}(I([1]b), \ldots, I([n]b)) = (I(b[1]), \ldots I(b[m]))$$

- *An interpretation $I$ of block-diagram $d$ is a* model *of $d$ iff*
$$\forall b \in B : I \text{ is a model of } b \text{ and } \forall (b[i], [j]b') \in C : I(b[i]) = I([j]b')$$

**Example 15** (Example 14 continued)**.** A block-diagram interpretation is presented in Figure 4.2. The interpretation is given by labeling all the inputs and all the outputs. For instance, $I([1]b_2)$ equals 4 and $I(b_3[1])$ equals 12. Moreover, this interpretation is a model of the block-diagram expressing that the input $2, 1$ produces the output 12.

Note that a block-diagram can have one or many models. The model presented in Example 15 is one among an infinity of possible ones.

### 4.2.3 Stream

Up to here, we built block-diagrams over arbitrary sets. Now, we consider the set of *streams*. A stream is an infinite discrete sequence of values possibly different at each time step.

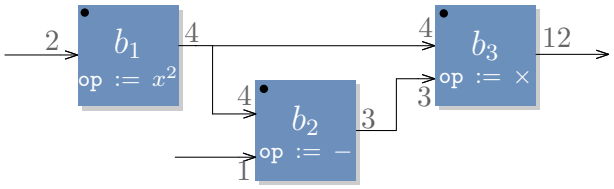



Figure 4.2: A block-diagram in $\mathtt{BD}(\mathbb{R})$ labeled with an interpretation

**Definition 4.2.6** (Stream)**.** *Let $D$ be a nonempty set. A stream $s$ over $D$ (also called a stream with domain $D$) is an infinite sequence of values from $D$. We name $s(t)$ the value at time $t$ for the stream $s$ ($t \in \mathbb{N}$). For any nonempty set $D$, we write $\mathbb{S}(D)$ the set of all the streams with domain $D$.*

We abbreviate streams using bracket notation. For instance the stream $s$ starting with the values 2, 4.5, and $-3$ (*i.e.*, $s(0) = 2$, $s(1) = 4.5$, $s(2) = -3$) is abbreviated in $[2, 4.5, -3, \ldots]$. In the following, it is important to remind that all the streams are *infinite*.

Considering block-diagrams over streams reveals two categories of blocks: *functional blocks*, and *temporal blocks*. Functional blocks can be computed independently at each time step, whereas temporal blocks have time dependencies. Functional blocks are introduced in Definition 4.2.7. Temporal blocks are blocks which are not functional blocks (*i.e.*, a block is either functional or temporal). We exhibit one block among all the temporal blocks: the *fby* block (cf. Definition 4.2.8). This block has two inputs and one output. The output at time zero is the value given by its first input at time zero. For the following times, the *fby* operator outputs its second input delayed by one time step.

**Definition 4.2.7** (Functional Block)**.** *Let $D$ be a nonempty set, and $b = (\mathtt{op}, n, m)$ in $\mathtt{Block}(\mathbb{S}(D))$. $b$ is a* functional block *iff $\exists f : D^n \to D^m$ such that $\forall s_1, \ldots, s_n, s'_1, \ldots, s'_m \in \mathbb{S}(D)$: $\mathtt{op}(s_1, \ldots, s_n) = (s'_1, \ldots, s'_m)$ implies the following:*

$$\forall t \in \mathbb{N}, f(s_1(t), \ldots, s_n(t)) = (s'_1(t), \ldots, s'_m(t))$$

**Definition 4.2.8** (Followed-by Block)**.** *Let $D$ be a nonempty set. The followed-by block over $D$ (written* fby*) is the block $(\mathtt{op}, 2, 1)$ in $\mathtt{Block}(\mathbb{S}(D))$ such that $\mathtt{op}$ is the function from $\mathbb{S}(D) \times \mathbb{S}(D)$ to $\mathbb{S}(D)$ where $\mathtt{op}(a, b) = c$, $c(0) = a(0)$, and $c(t) = b(t-1)$, for all $t > 0$.*

**Example 16.** Figure 4.3 shows a block-diagram $d$ over real-number streams: $d \in \mathtt{BD}(\mathbb{S}(\mathbb{R}))$. $d$ has no input and no output. $d$ contains 5 functional blocks: 0, 0.1, 0.9, $+$, and $\times$. Blocks 0, 0.1 and 0.9 use constant operators (*i.e.*, $\forall t \in \mathbb{N} : 0.9(t) = 0.9$). Blocks $+$ (resp. $\times$) with real-number input streams $a$, $b$ and real-number output stream $c$ is such that $c(t) = a(t) + b(t)$ (resp. $c(t) = a(t) \times b(t)$). $d$ contains one temporal block: the *fby* block (note that temporal blocks are hatched).

Block-diagrams over streams are used for programming real-time applications. In this context we name *execution trace* or simply *trace* a model of a block-diagram. A cycle in a

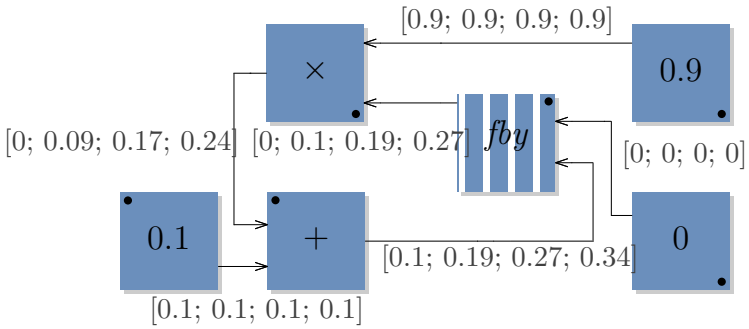



Figure 4.3: A block-diagram over streams from $\mathtt{BD}(\mathbb{S}(\mathbb{R}))$. In brackets, the first values of the model for $t = 0, 1, 2, 3$.

block-diagram is equivalent to a loop in a classic programming language. In practice, in order to be *runnable* (*i.e.*, to compute a trace in real-time) a block-diagram over streams needs to satisfy two properties [85]: no value must be depending on future values (called the causality property); infinite computation in cycle must be avoided (any cycle must contain temporal blocks to avoid infinite computation at each time step).

In our contribution we only allow to use the *fby* block as temporal block. Under this condition, this implies that for any runnable block-diagram over streams each cycle contains at least one *fby* block. From now on, we only consider block-diagram verifying this statement. We will see in Section 4.6 that this restriction is not poor and that the *fby* block allows to represent many other temporal blocks.

**Example 17** (Example 18 continued)**.** Block-diagram $d$ over real-number streams in Figure 4.3 contains one cycle which contains one *fby* block. This block-diagram admits only one model/trace. Indeed, using the three constant blocks 0, 0.1 and 0.9 fixes all the values in the cycle. Values for the 4 first time steps of the model are attached to each connector (note the delay due to the *fby* block). Values for the 21 first time steps are presented in Figure 4.4. Note the delay between the output of the block + and the output of the *fby* block: the height of the circle corresponds to the height of the square at the previous time step.

## 4.3 Stream Over-Approximation Problem

Block-diagrams over streams can express the semantics of real-time programs. In such cases, the programmer could be interested in the verification of some properties of his/her program. These properties can concern outputs or internal streams (*i.e.*, outputs or local variables). We propose here a logical constraint model for the following problem: determine bounds of the streams of a block-diagram.

### 4.3.1 Temporal and Interval Abstractions

We illustrate the problematic on our running example. Figure 4.4 presents the first 21 values for the output streams of blocks $\times$, $+$, and *fby* model of our running example from Figure 4.3. For the first 21 time steps, one can see that the values are strictly increasing (*i.e.*, streams are strictly increasing) between 0 and 1. Furthermore the same observation is still correct for the first 100, $1,000$, $1,000,000$, and more, time steps (in our example, model streams are "infinitely" strictly increasing). Clearly, the greedy algorithm running the block-diagram time by time and gathering the accessible states (a state is a tuple composed of the values of all the streams at one time step) until convergence (*i.e.*, no new state is reached) may not halt. Furthermore, a block-diagram can admit an infinite uncountable set of models/traces. Thus, there is no hope to run all these traces for gathering all the reachable states. In this context, Abstract Interpretation [54, 53] offers a great variety of tools for over-approximating traces of programs. It relies on abstractions of the program traces. The set of all the possible program traces is undecidable in the general case. In Abstract Interpretation, they are represented by an abstract element, easier to compute, which must both include all the program traces and be reasonably easy to compute. One of the first examples of such abstraction is the interval abstract domain [53]. While finding one over-approximation of the traces is easy (*i.e.*, returning $[-\infty, +\infty]$) the objective is to find over-approximations with good quality (*i.e.*, as small as possible intervals). Following paragraphs formally introduce the over-approximation problem with the over-approximation quality comparator.

**Problem Definition.** Let $D$ be a nonempty set and $d$ be a block-diagram in $\mathtt{BD}(\mathbb{S}(D))$. Associate to each block input/output $s$ in $d$ a subset $S$ of $D$ s.t. for each model/trace $I$ of $d$ and for each time step $t$ in $\mathbb{N}$ the value $s(t)$ is in $S$. $S$ is called an over-approximation of $s$ in $d$.

**Over-Approximation Quality.** Let $D$ be a nonempty set, $d$ be a block-diagram in $\mathtt{BD}(\mathbb{S}(D))$, $s$ be a block input/output in $d$, and $S, S'$ subsets of $D$ be two over-approximations of $s$ in $d$. If $S \subseteq S'$ then the over-approximation $S$ is preferred to the over-approximation $S'$.

**Example 18** (Example 17 continued)**.** Interval $[0, 1]$ contains all the values taken by the streams model of the outputs for the blocks $+$, $\times$ and *fby* for the first 21 time steps in the block-diagram in Figure 4.3. Interval $[0.1, 0.9]$ is a better over-approximation than $[0, 1]$ for the output of the block $+$ for the 21 first time steps.

We introduce the temporal abstraction of streams in Definition 4.3.1. The temporal abstraction of a stream returns the set of all values taken by this stream. This set (and any superset) is called an over-approximation of the stream. As said previously, the size of this

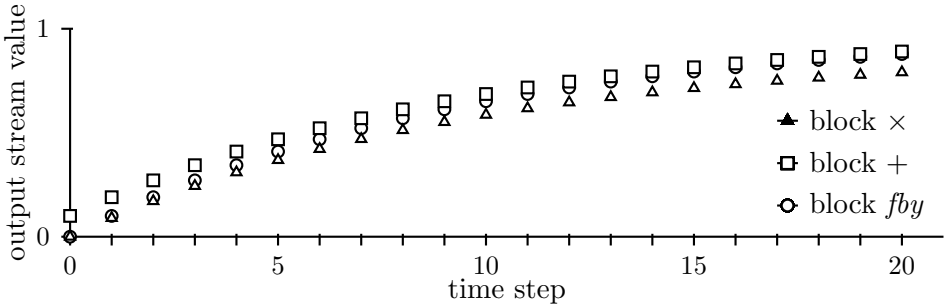



Figure 4.4: Values of the streams model of the block-diagram in Figure 4.3 for the outputs of blocks ×, +, and *fby* for the 21 first time steps.

set may be infinite, discontinuous and even uncountable (*i.e.*, a representation in extension is thus not possible). Thereby, given a stream we consider an interval superset of the temporal abstraction for representing this stream (*i.e.*, this corresponds to the use of the interval abstract domain in Abstract Interpretation [53]). The best over-approximation (in the intervals) of a stream, is the smallest interval containing its temporal abstraction.

**Definition 4.3.1** (Temporal Abstraction)**.** *The temporal abstraction of a stream $s$, written $\dot{s}$, is the set of all its values. Any superset of $\dot{s}$ is called an* over-approximation *of $s$ and $\dot{s}$ is the smallest over-approximation of $s$.*

$$\dot{s} = \bigcup_{t \in \mathbb{N}} s(t)$$

For each interval $I$, we write $\lceil I \rceil$ its upper bound and $\lfloor I \rfloor$ its lower bound. In the following, we assume that $D$ is a totally ordered set and we write $\mathbb{I}(D)$ the set of all the intervals over $D$. Furthermore, $\overline{D}$ is called the extended set of $D$ and it is equal to the union of $D$ and its limits (*e.g.*, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ and $\mathbb{I}(\overline{\mathbb{R}})$ is the set containing all the intervals with finite and infinite bounds). Finally, given $A \subseteq D$ we write $[A]$ the smallest interval in $\mathbb{I}(D)$ containing $A$.

### 4.3.2 Model in Constraint Programming

Constraint programming (`CP` for short) is a declarative programming paradigm, in which a program consists of a list of variables (each one declared with a domain) together with a list of constraints over these variables. Firstly, we do constraint programming modelling with variables domains over streams. Secondly, we focus on interval constraint programming [86], *i.e.*, constraint programming with variable domains over set of intervals.

**Definition 4.3.2** (Constraint Satisfaction Problem)**.** *Let $E$ be a nonempty set. A Constraint Satisfaction Problem (*CSP *for short) is a tuple $\mathcal{P} = (X, D, C)$ such that $X$ is a finite set of variables; each variable $x$ in $X$ is associated with a domain $D_x$ subset of $E$; $D$*

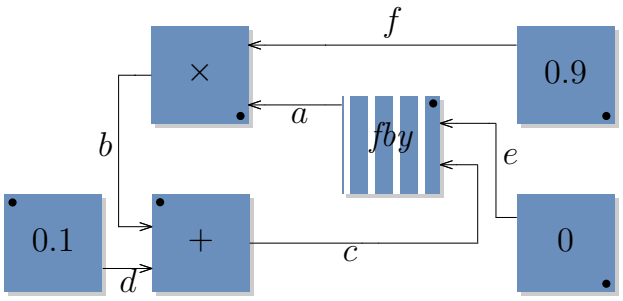



Figure 4.5: A block-diagram over streams from $\mathtt{BD}(\mathbb{S}(\mathbb{R}))$ with connectors labelled by variables

$$
\begin{array}{l}
\mathrm{a}\,,\mathrm{b}\,,\mathrm{c}\,,\mathrm{d}\,,\mathrm{e}\,,\mathrm{f} \in \mathbb{S}(\mathbb{R}) \\
\\
\mathrm{a} \;=\; \mathit{fby}(\,\mathrm{e}\,,\mathrm{c}\,) \\
\mathrm{b} \;=\; \times(\,\mathrm{a}\,,\mathrm{f}\,) \\
\mathrm{c} \;=\; +(\,\mathrm{b}\,,\mathrm{d}\,) \\
\mathrm{d} \;=\; 0.1 \\
\mathrm{e} \;=\; 0 \\
\mathrm{f} \;=\; 0.9
\end{array}
$$

Figure 4.6: Naive constraint model for the block-diagram in Fig 4.5

$$
\begin{array}{l}
\mathrm{a}\,,\mathrm{b}\,,\mathrm{c}\,,\mathrm{d}\,,\mathrm{e}\,,\mathrm{f} \in \mathbb{I}(\overline{\mathbb{R}}) \\
\\
\mathrm{a} \;=\; [\mathit{fby}](\,\mathrm{e}\,,\mathrm{c}\,) \\
\mathrm{b} \;=\; [\times](\,\mathrm{a}\,,\mathrm{f}\,) \\
\mathrm{c} \;=\; [+](\,\mathrm{b}\,,\mathrm{d}\,) \\
\mathrm{d} \;=\; [0.1] \\
\mathrm{e} \;=\; [0] \\
\mathrm{f} \;=\; [0.9]
\end{array}
$$

Figure 4.7: Medium constraint model for the block-diagram in Fig 4.5

*is the set of all the domains associated to the variables in $X$; $C$ is a set of constraints over variables from $X$. A constraint is defined over a set of variables $x_1, \ldots, x_k$ from $X$ with $k \in \mathbb{N}$ and is a subset of $D_{x_1} \times \ldots \times D_{x_k}$. A valuation $v$ of $\mathcal{P}$ is a map from $X' \subseteq X$ to $D$ s.t. $v(x) \in D_x$ for all $x \in X'$. A valuation $v$ satisfies a constraint $c \subseteq D_{x_1} \times \ldots \times D_{x_k}$ iff $(v(x_1), \ldots, v(x_k)) \in c$. Finally, a valuation satisfies $\mathcal{P}$ iff it satisfies all the constraints in $\mathcal{P}$.*

Definition 4.3.2 introduces Constraint Satisfaction Problems. We propose to model as a CSP the stream over-approximation problem. Block-diagrams compute outputs from inputs. To determine over-approximations of the streams in a block-diagram $(B, C)$ in $\mathtt{BD}(\mathbb{S}(D))$, we associate to each input and to each output from the blocks in $B$ a *variable* with domain $\mathbb{S}(D)$. Then, for each block in $B$ we consider its operator as a *constraint* linking the block outputs to the block inputs. Furthermore, for each connector in $C$, we add a constraint to ensure the equality of its streams. We name *naive model* this model using variables over streams. Example 19 presents the naive model on our running example.

**Example 19** (Example 18 continued)**.** Figure 4.5 contains the same block-diagram as in Example 18 with constraint variables associated to the inputs and the outputs. Note that in our example, variables have been unified per connectors (*e.g.*, $[1]+ = *[1] = \mathrm{b}$).

About the constraint programming model, the block with the operator + computes $c$ as a function of $b$ and $d$, yielding the constraint: $c = b + d$. Figure 4.6 shows the constraint model over streams for our example.

We recall that the over-approximation problem presented in the previous section asks for over-approximations of all the traces of the block-diagram. Since one solution of the naive model corresponds to one trace of the block-diagram, one must find all the solutions of the naive model to solve the over-approximation problem. In the previous section we motivated the use of over-approximations in the intervals for representing set of traces. We now present a second model with variables over intervals for solving the over-approximation problem.

This model, called *medium model*, is derived from the *naive model*. It consists in: 1) the same variables where domains are over intervals instead of streams (*i.e.*, over $\mathbb{I}(\overline{D})$ instead of $\mathbb{S}(D)$); 2) the same signatures of constraints where the operators over streams are replaced by their corresponding constraint for interval propagation. Interval propagation combines various technics from interval arithmetic, interval constraint propagation, domain filtering with partial consistency algorithms. Note that these extensions are not trivial and continue to motivate researchers (see [86, 87, 88]). In the following, we will particularly use interval arithmetic and interval (constraint) propagation.

**Example 20** (Interval Arithmetic)**.** Instances of interval computation:

$$\begin{array}{rclrcl} [2,6]+[-1,3] & = & [1,9] & [2,6]\times[-1,3] & = & [-6,18] \\ [2,6]-[-1,3] & = & [-1,7] & [-1,3]\times[-1,3] & = & [-3,9] \\ [2,6]-[2,6] & = & [-4,4] & [-1,3]^2 & = & [0,9] \end{array}$$

Note that some properties in real-number arithmetic are not true in interval arithmetic. Examples above illustrate that in general $A - A \neq [0,0]$ and $A^2 \neq A \times A$.

**Definition 4.3.3** (Interval Extension Function)**.** *Let $D$ be a nonempty set and $f$ be a function from $\mathbb{S}(D)^n$ to $\mathbb{S}(D)^m$ with $n, m \in \mathbb{N}$. An interval extension function of $f$, is a function $[f]$ from $(\mathbb{I}(D))^n$ to $(\mathbb{I}(D))^m$ such that $[f](X_1, ..., X_n) = Y_1, ..., Y_m$ where $Y_i = [\{\dot{y_i} \mid \exists x_j \in X_j, y_1, ..., y_m = f(x_1, ..., x_n)\}]$.*

*Interval arithmetic* received big interest since Moore [89] and the developments of interval analysis. We focus on interval arithmetic with interval extension of real-valued functions. Interval arithmetic concerns how classical functions from real-number arithmetic operates on intervals (see Example 20). We propose Definition 4.3.3 for transposing function over streams to interval functions which extends the definition from [89] for extending real-number functions to interval functions. Table 4.1 presents standard arithmetic functions over real-numbers next to their corresponding stream functions and interval extension functions. When it is not ambiguous (*i.e.*, in the context of intervals) we omit the brackets over the function names in order to keep the expressions simpler.

| Real Function | Stream Function | Interval Extension Function |
|---|---|---|
| $a, b \mapsto a + b$ | $a, b \mapsto c$, s.t.<br>$c(t) = a(t) + b(t), \forall n \in \mathbb{N}$ | $[a_1, a_2], [b_1, b_2] \mapsto [a_1 + b_1, a_2 + b_2]$ |
| $a, b \mapsto a - b$ | $a, b \mapsto c$, s.t.<br>$c(t) = a(t) - b(t), \forall n \in \mathbb{N}$ | $[a_1, a_2], [b_1, b_2] \mapsto [a_1 - b_2, a_2 - b_1]$ |
| $a, b \mapsto a \times b$ | $a, b \mapsto c$, s.t.<br>$c(t) = a(t) \times b(t), \forall n \in \mathbb{N}$ | $[a_1, a_2], [b_1, b_2] \mapsto [c_1, c_2]$ s.t.<br>$c_1 = min(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2)$<br>$c_2 = max(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2)$ |
| $a \mapsto a^2$ | $a \mapsto c$, s.t.<br>$c(t) = a(t)^2, \forall n \in \mathbb{N}$ | $[a, b] \mapsto [c, max(a^2, b^2)]$ s.t.<br>$c = 0$, if $a \leq 0 \leq b$<br>$c = min(a^2, b^2)$, otherwise |

Table 4.1: Real-number functions, stream functions, and interval extension functions for the addition, the subtraction, the multiplication and the square functions

The interval extension function of an operator is not unique but the functions with smallest images will by preferred (*i.e.*, the function always returning $\overline{D}$ is a universal interval extension function).

*Constraint propagation* is one of the key ingredient for CSP resolution [86]. This consists in explicitly removing values in some variables domains which cannot satisfy the CSP, while preserving all the solutions. A function performing such operation over one constraint is called a *propagator* (cf. Definition 4.3.4). For instance consider the constraint $a = b + c$ over intervals. The function $f(A, B, C) = (A \cap (B + C), B \cap (A - C), C \cap (A - B))$ is a propagator for this constraint. If the domains for the variables $a$, $b$, and $c$ are respectively $[-\infty, 4]$, $[-1, 3]$, and $[0, +\infty]$ then the propagator $f$ reduces the domains for the variable $a$, $b$, and $c$ to respectively $[-1, 4]$, $[-1, 3]$ and $[0, 5]$.

**Definition 4.3.4** (Constraint Propagator)**.** *Let $(X, D, C)$ be a* CSP *with $X = \{x_1, \ldots, x_n\}$, and let $c$ be a constraint in $C$ defined over the set of variables $X' \subseteq X$. A propagator $f$ for the constraint $c$ is a function from $\mathcal{P}(D)$ to $\mathcal{P}(D)$ such that $f(D'_{x_1}, \ldots, D'_{x_n}) = D''_{x_1}, \ldots, D''_{x_n}$ with*

- *for all $x \in X \setminus X' : D''_x = D'_x$*
- *for all $x \in X' : D''_x \subseteq D'_x$*
- *for all valuations $v$ of $x_1, \ldots, x_n$ in $D'_{x_1}, \ldots, D'_{x_n}$: if $v$ satisfies $c$, then $v(x_i) \in D''_{x_i}$ for all $x_i \in X'$.*

Figure 4.7 shows the medium constraint model for the block-diagram in Figure 4.5 constructed from its naive constraint model over streams in Figure 4.6. Solving this constraint model computes an interval over-approximation of each stream, provided that the interval extensions of the functions are correct. Therefore, this translation of block-diagrams into a constraint problem allows to compute hulls (over-approximations) of the streams.

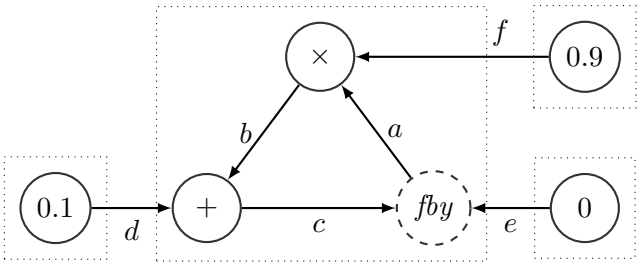



Figure 4.8: Dependency graph of the block-diagram in Figure 4.5 where strongly connected components are surrounded with dashed lines

## 4.4 The `real-time-loop` constraint

Since the stream domains can be infinite and the constraint network can contain *cycles*, classic constraint solvers may poorly reduce the domains when using the medium model. Consider our running example given in Figure 4.5 with its corresponding medium model given in Figure 4.7. Starting with domains $\mathbb{I}(\overline{\mathbb{R}})$ (*i.e.*, with $[-\infty, +\infty]$ in the domains) the domains for the triple $(a, b, c)$ of variables is glued to its whole domain $\mathbb{I}(\overline{\mathbb{R}})$ and interval propagators fail to reduce domains in order to compute smaller over-approximations. Moreover, assume that the stream domains are bounded: $D = [-d, d]$ with $d \in \mathbb{R}$. In such cases interval propagation will contract the domains blocks after blocks (*i.e.*, constraint after constraint). However, the convergence may appear after a huge number of interval propagations. In order to reach the gap to better over-approximation in less time we introduce a new constraint: the `real-time-loop` constraint.

### 4.4.1 Definition

The `real-time-loop` constraint will model cycles[2] in block-diagrams. A cycle in a block-diagram corresponds to a directed cycle in the directed graph representing it. A cycle is a sub block-diagram in a block-diagram. The `real-time-loop` constraint takes three arguments: the cycle itself as a block-diagram/list of constraints, the cycle inputs as a vector of variables and the cycle outputs as a vector of variables. Let $d$ be a block-diagram cycle, `inputs` be its inputs, and `outputs` be its outputs, we instantiate the `real-time-loop` constraint as: `real-time-loop`(d,`inputs`,`outputs`) . An interpretation satisfies a `real-time-loop` constraint if and only if it satisfies the list of constraints (*i.e.*, all the constraints). According to this new constraint, we propose two propagators in the following sections. The first one propagates from input domains to output domains and the second one do the opposite way. Example 21 illustrates the `real-time-loop` constraint on our running example.

[2]For reading facilities, we simply write *cycle* instead of *directed cycle* while working with directed structures such as block-diagrams.

```
a , b , c , d , e , f ∈ 𝕀(ℝ̄)

d = 0.1
e = 0
f = 0.9
real-time-loop([a=fby(e,c); b=×(a,f); c=+(b,d)], [d;e;f], [])
```

Figure 4.9: Optimized model of the block-diagram in Figure 4.5

**Example 21** (Example 19 continued)**.** Running example in Figure 4.5 has one cycle which contains three blocks ($+$, $\times$ and *fby*), three inputs ($d$, $e$, and $f$), and no outputs. This cycle is modelled with the `real-time-loop` constraints as follows: `real-time-loop` $([a = fby(e, c); b = \times(a, f); c = +(b, d)], \{d, e, f\}, \emptyset)$.

### 4.4.2 Optimized Model

This section describes how we exploit the structure of block-diagrams to improve the precision of the over-approximations using our `real-time-loop` constraint in an *optimized model*. Even if there is a thin syntactical difference between the medium model and this optimized model, there is a big gap in terms of deduction power.

**Definition 4.4.1** (Dependency Graph)**.** *Let $E$ be a nonempty set, and $d = (B, C)$ be a block-diagram in* `BD`$(E)$*. The dependency graph of $d$ is the directed graph $G = (V, A)$ in which each* node *of $V$ corresponds to a different block from $B$ such that $|V| = |B|$ and each* arc *of $A$ corresponds to a different connector from $C$ such that $|A| = |C|$.*

From a constraint programming point of view, these graphs are the constraints dependency graphs (where nodes are the CSP constraints), except that the arcs are directed by the dependencies implied by the blocks. Figure 4.8 draws the dependency graph of the block-diagram in Figure 4.5. Again, temporal block nodes are hatched. The optimized model is derived from the first one presented in section 4.3.2. Note that each strongly connected component (*i.e.*, set of nodes such that it exists a path between any two nodes from this set) in the dependency graphs is related to a loop in the block-diagram. Thereby, regarding the dependency graph of the block-diagram, we compute its strongly connected components and we replace for each one all its corresponding constraints in the medium model by one `real-time-loop` constraint taking the strongly connected component as argument. Figure 4.9 models the block-diagram in Figure 4.5 using the `real-time-loop` constraint. Note that in this model the `real-time-loop` constraint has three variables as inputs and none as outputs.

### 4.4.3 Inputs to outputs propagator

We now present how to propagate the `real-time-loop` constraints from inputs to outputs: according to over-approximations of the inputs of the loop, we want to determine over-approximations for the outputs of the loop. Remember that the block-diagram is evaluated over infinite discrete time. Given a cycle/loop, we extract a *transfer function* for this loop and then, we consider the *interval extension* function of this transfer function in order to find over-approximations.

**Definition 4.4.2** (Loop Transfer Function)**.** *Let $d$ be a cycle block-diagram in $\mathtt{BD}(\mathbb{S}(D))$ and $X = \{x_1, \ldots, x_k\}$ be a set of blocks inputs or outputs from $d$ called* argument*. $F : D^k \mapsto D^k$ is a loop transfer function of $d$ for argument $X$, iff for all $I$ model of $d$ and for all $t$ in $\mathbb{N}$: $F(I(x_1)(t), \ldots, I(x_k)(t)) = I(x_1)(t+1), \ldots, I(x_k)(t+1)$*

Given a set of block inputs or outputs, a loop transfer function computes values at the next time according to values at a given time. Real-time languages must ensure the causality property [85] (*i.e.*, it must not exist a stream computing its values according to future values). Due to this property, it is clear that each cycle block-diagram admits at least one loop transfer function and even admits at least one loop transfer function with an argument of minimal size. This problem can be reduced to a "covering graph problem". Let $d$ be a cycle block-diagram, $G = (V, A)$ be the dependency graph of $d$, and $S \subseteq V$ be a set of vertices. The set $S'$ such that $S' \supseteq S$, for all $s \in S'$ all its predecessors are in $S'$, and $S'$ is minimal, is named the *cover* of $G$ by $S$. Furthermore, we say that $S$ is a *causal set* of $G$ if the cover of $G$ by $S$ equals to $V$. Thus, finding a loop transfer function with an argument of minimal size can be reduced to finding a minimal causal set and then performing a breadth-first search from this set for constructing the transfer function. We propose a greedy algorithm, Algorithm 2, for computing a minimal causal set of a dependency graph. It enumerates the subsets of $V$ by starting from the subsets with minimal size and stops when it has found a causal set. In our benchmark presented in Table 4.4, we can see that this greedy algorithm does not run out-of-time (*i.e.*, in practice in our benchmark it does not enumerate all the subsets of $V$ but only a small amount). Finally, we use the Definition 4.3.3 to get a *loop transfer function extended to the intervals* (in the following, we simply call it a *loop transfer function* too). Once we get this function, we want to over-approximate associated streams. Proposition 1 allows to do so by finding stable intervals such as defined in Definition 4.4.3. An example is given below.

**Definition 4.4.3** (Interval Stability)**.** *Let $D$ be a non-empty set, $F$ be an interval function with arity $n \in \mathbb{N}$, and $S_1, \ldots, S_n$ be $n$ intervals from $\mathbb{I}(\overline{D})$. We say that $S_1, \ldots, S_n$ is stable by $F$ iff $F(S_1, \ldots, S_n) = S'_1, \ldots, S'_n$ s.t. $S'_i \subseteq S_i$ for all $1 \leq i \leq n$.*

```
1: function MINIMALCAUSALSET(G : Graph) return Set<Vertex>
2:     stack : Set<Vertex>
3:     cover : Set<Vertex>
4:     V ← G.getVertices() : Set<Vertex>
5:
6:     # Look for the first subset of V which is a causal set
7:     for each A ⊆ V enumerated by increasing size do
8:         stack ← A
9:         cover ← A
10:
11:        # Computing cover of G by A
12:        while not(stack.isEmpty()) do
13:            v ← stack.pop()
14:            for each v' ∈ G.getSuccessors(v) do
15:                if v' ∉ cover and G.getPredecessors(v') ⊆ cover then
16:                    stack.push(v')
17:                    cover.add(v')
18:                end if
19:            end for each
20:        end while
21:
22:        # Check if it is a causal set
23:        if cover = V then
24:            return A
25:        end if
26:    end for each
27: end function
```

Algorithm 2: Compute Minimal Causal Set of a Dependency Graph

**Example 22** (Example 21 continued)**.** Let $f$ and $g$ be two functions from $\mathbb{R}$ to $\mathbb{R}$ such that $f(y) = \mathit{fby}(0, y \times 0.9 + 0.1)$ and $g(y) = 0.9 \times \mathit{fby}(0, y) + 0.1$ for all $y \in \mathbb{R}$. Remind from Figure 4.5 that the symbol $a$ stands for the *fby* block output and the $\times$ block first input. We have that $f$ with argument $\{a\}$ and $g$ with argument $\{c\}$ are two loop transfer functions for the cycle in our block-diagram running example. and that the symbol $c$ stands for the $+$ block output and the *fby* block second input. Thus, for any model $I$ of the block-diagram and for all time step $t$ in $\mathbb{N}$, the value associated to $a$ (resp. $c$) by $I$ at time $t+1$ corresponds to the image by $f$ (resp. $g$) of the value associated to $a$ (resp. $c$) by $I$ at time $t$ (*i.e.*, it holds that $I(a)(t+1) = f(I(a)(t))$ and $I(c)(t+1) = g(I(c)(t))$).

Let $F$ and $G$ be two functions from $\mathbb{I}(\mathbb{R})$ to $\mathbb{I}(\mathbb{R})$ such that $F(Y) = \mathit{fby}([0], Y \times [0.9] + [0.1])$ and $G(Y) = [0.9] \times \mathit{fby}([0], Y) + [0.1]$ (here in the context of intervals the function "$\times$", "$+$", and "*fby*" are not the real valued functions but their respective interval extension functions). Function $F$ extends $f$ to the intervals and function $G$ extends $g$ to the intervals. We have that $F$ with argument $\{a\}$ and $G$ with argument $\{c\}$ are two loop transfer functions (extended to the intervals) for the cycle in our block-diagram running example.

Considering the loop transfer function $F$. Intervals $[0;1]$, $[-1;1]$ and $[-4;3]$ are stable by $F$. (the images are respectively $[0;1]$, $[-0.8;1]$ and $[-3.5;2.8]$). Thus, by Proposition 1 all these intervals are valid over-approximations for stream $c$ (*i.e.*, the argument of $F$). On the contrary intervals $\emptyset$ and $[0,0]$ are not stable (their images are respectively $[0;0]$ and $[0;0.1]$). We conclude that $\emptyset$ and $[0,0]$ are not valid over-approximations for stream $c$.

One of our main contributions is Algorithm 3. We propose a method inspired by abstract interpretation techniques viewed as a constraint program to determine stable sets of intervals. Proposition 2 states the correctness of the algorithm. Note that this algorithm may not systematically return the minimal over-approximation, but in practice it gives acceptable ones (see experiments in Section 4.6). This algorithm starts by associating each argument element of the function to a search space bounded by the intervals $min[i]$ and $max[i]$ which are respectively initialized with the empty set and the extended set of the considered domain. Then, at each iteration $current[i]$ is selected such that it contains $min[i]$ and it is contained in $max[i]$ (*i.e.*, $min[i] \subseteq current[i] \subseteq max[i]$ is an invariant of the loop). Also, the variable *state* takes its values between "Increasing" and "Decreasing" and is initialized to "Increasing". It switches from *increasing* to *decreasing* when the interval $current[i]$ is stable by the transfer function and switches from *decreasing* to *increasing* when the contrary occurs. Finally, functions SELECTINTERVALBETWEEN and CONTINUELOOPING are heuristics (resp. *selection heuristic* and *looping heuristic*).

**Proposition 1.** *Let $D$ be a nonempty set, $d = (B, C)$ be a block-diagram in $\mathtt{BD}(\mathbb{S}(D))$ and $F$ be a loop transfer function (extended to intervals) of $d$ with argument $X$ of size $k$. If $S$ in $\mathbb{I}(\overline{D})^k$ is stable by $F$ then, $S$ is an over-approximation of the elements in $X$.*

**Proposition 2.** *[Algorithm 3 Correctness] Let $(u_n)_{n\in\mathbb{N}}$, $(v_n)_{n\in\mathbb{N}}$, and $(w_n)_{n\in\mathbb{N}}$ be the sequences of values taken respectively by the variables "min", "current", and "max" at each evaluation of the loop condition (line 8) during an execution of Algorithm 3 over a function $F$. The following statements hold:*

1. *$(u_n)$ is increasing and $(w_n)$ is decreasing*
2. *for all $n \in \mathbb{N}$: $u_n \subseteq v_n \subseteq w_n$*
3. *for all $n \in \mathbb{N}$: $w_n$ is stable by $F$.*

*Proof.* Let $(u_n)_{n\in\mathbb{N}}$, $(v_n)_{n\in\mathbb{N}}$, and $(w_n)_{n\in\mathbb{N}}$ be the sequences of values taken respectively by the variables "*min*", "*current*", and "*max*" at each evaluation of the loop condition (line 8) during an execution of Algorithm 3 with a function $F$ from $\mathbb{I}(D)^k$ to $\mathbb{I}(D)^k$ ($k \in \mathbb{N}$). Note that values for $u_n$, $v_n$, and $w_n$ are in $\mathbb{I}(D)^k$. Let $n \in \mathbb{N}$ and $i \in \{1, \ldots, k\}$ we write $u_n[i]$, $v_n[i]$, and $w_n[i]$ the $i$th interval in $u_n$, $v_n$, and $w_n$ respectively. Then we have $u_n[i]$, $v_n[i]$, and $w_n[i]$ belonging to $\mathbb{I}(D)$.

```
 1: function overApproximation(f : 𝕀(D̄)^k → 𝕀(D̄)^k) return List<𝕀(D̄)>
 2:     state ← "Increasing"
 3:     current, min, max, image : List<𝕀(D̄)>
 4:     for each i from 1 to k do
 5:         current[i] ← ∅, min[i] ← ∅, max[i] ← D̄
 6:     end for each
 7:
 8:     while continueLooping(current, min, max) do
 9:         image ← f(current)
10:         switch ← false
11:
12:         if (state = "Increasing" and image ⊆ current) then
13:             state ← "Decreasing", switch ← true
14:         else if state = "Decreasing" and image ⊈ current then
15:             state ← "Increasing", switch ← true
16:         end if
17:
18:         for each i from 1 to k do
19:             if switch and state = "Increasing" then
20:                 min[i] ← current[i]
21:             else if state = "Decreasing" then
22:                 max[i] ← current[i]
23:             end if
24:
25:             current[i] = current[i] ∪ image[i]
26:
27:             if state = "Increasing" then
28:                 random ← selectIntervalBetween(current[i], max[i])
29:             else
30:                 random ← selectIntervalBetween(min[i], current[i])
31:             end if
32:
33:             current[i] ← random
34:         end for each
35:     end while
36:
37:     return max
38: end function
```

Algorithm 3: Over-approximation random search function

Proof for Statements 2 and 3 are obtained by induction on the number of evaluations of the loop condition. We first check the validity of both statements at the first evaluation of the loop condition (line 8). We have that $u_0[i] = v_0[i] = \emptyset$, and $w_0[i] = \overline{D}$ for all $i \in \{1, \ldots, k\}$. Clearly, $u_0[i] \subseteq v_0[i] \subseteq w_0[i]$ for all $i \in \{1, \ldots, k\}$. This implies that $u_0 \subseteq v_0 \subseteq w_0$ (Statement 2). Furthermore $w_0$ equals to $\overline{D}^k$ makes $F(w_0) \cup w_0 = \overline{D}^k = w_0$ which means that $w_0$ is stable by $F$ (Statement 3). Assume now that Statements 2 and 3 holds for the $n$th evaluation of the loop condition. We prove that both statements are

| Variable \ Iteration | 1 | 2 | 3 | 4 | 5 | 6 | ... | $n$ |
|---|---|---|---|---|---|---|---|---|
| *state* | Increasing | Increasing | Decreasing | Decreasing | Decreasing | Increasing | ... | Any |
| *current* | $\emptyset$ | $[-10; 100]$ | $[-3; 6]$ | $[0; 2]$ | $[0; 0.6]$ | $[0; 1.2]$ | ... | $[0; 1]$ |
| *image* | $[0; 0]$ | $[-8.9; 90.1]$ | $[-2.6; 5.5]$ | $[0.1; 1.9]$ | $[0.1; 0.64]$ | $[0.1; 1.18]$ | ... | $[0; 1]$ |
| *union* | $[0; 0]$ | $[-10; 100]$ | $[-3; 6]$ | $[0; 2]$ | $[0; 0.64]$ | $[0; 1.18]$ | ... | $[0; 1]$ |
| *switch* | No | Yes | No | No | Yes | Yes | ... | Yes |
| *min* | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $[0; 0.6]$ | $[0; 0.6]$ | ... | $[0; 1]$ |
| *max* | $[-\infty; +\infty]$ | $[-10; 100]$ | $[-3; 6]$ | $[0; 2]$ | $[0; 2]$ | $[0; 1.2]$ | ... | $[0; 1]$ |
| *random* | $[-10; 100]$ | $[-3; 6]$ | $[0; 2]$ | $[0; 0.6]$ | $[0; 1.2]$ | $[0; 0.9]$ | ... | $[0; 1]$ |

Table 4.2: A trace table of Algorithm 3 for the transfer function $F$.

still correct for the $n+1$th iteration. There are 4 cases depending on the variable *states* and *current* (*i.e.*, $v_n$):

1. *state* = "Increasing" and $F(v_n) \subseteq v_n$
2. *state* = "Increasing" and $F(v_n) \not\subseteq v_n$
3. *state* = "Decreasing" and $F(v_n) \subseteq v_n$
4. *state* = "Decreasing" and $F(v_n) \not\subseteq v_n$

Consider the first case. Condition in line 12 is true. This sets the variable *state* to "Decreasing" and the variable *switch* to "true". Next, in the for statement only the condition in line 21 is true. Thus, for all $i \in \{1, \ldots, k\}$: $max[i]$ is updated to $current[i]$ (*i.e.*, $w_{n+1}[i] = v(n)[i]$); $current[i]$ is updated to an interval between $min[i]$ and its current value (*i.e.*, $u_n[i] \subseteq v_{n+1}[i] \subseteq v_n[i]$) and such interval exists by the inductive hypothesis $v_n \subseteq u_n \subseteq w_n$; and $min[i]$ is unchanged (*i.e.*, $u_{n+1}[i] = u_n[i]$). Finally we obtain by aggregation that $u_{n+1}[i] = u_n[i] \subseteq v_{n+1}[i] \subseteq v_n[i] = w_{n+1}[i]$ (Statement 2). Moreover, the value $v_n$ set to $w_{n+1}$ (cf. $max[i] \leftarrow current[i]$, for all $i \in \{1, \ldots, k\}$) verify $F(v_n) \cup v_n = v_n$ in the considered case. Thus $w_{n+1}$ is stable by $F$ (statement 3). Proofs for cases 2, 3, and 4 are similar.

Proof for Statement 1. Let $i$ be in $\{1, \ldots, k\}$. Note that $min[i]$ (*i.e.*, $u_n[i]$) is only updated at line 20 and that $max[i]$ (*i.e.*, $w_n[i]$) is only updated at line 22. Both are updated with the value of $current[i]$ (*i.e.*, $v_n[i]$). Thus, we get that for all $n \in \mathbb{N}$: $u_{n+1}[i] = v_n[i]$ or $u_{n+1}[i] = u_n[i]$; and $w_{n+1}[i] = v_n[i]$ or $w_{n+1}[i] = w_n[i]$. We get by statement 2 that $u_n[i] \subseteq u_{n+1}[i]$ and $w_{n+1}[i] \subseteq w_n[i]$ and this is correct for all $i$ in $\{1, \ldots, n\}$. We conclude that $u_n \subseteq u_{n+1}$ and that $w_{n+1} \subseteq w_n$, *i.e.*, $u_n$ is increasing and $w_n$ is decreasing. □

**Example 23** (Example 22 continued)**.** Table 4.2 details a trace of Algorithm 3 for the transfer function $F$ in Example 22. Each column corresponds to one iteration of the "while" loop. Each line gives the values of the variables at the end of each iteration, except for *current* which contains the value when starting the iteration.

### 4.4.4 Outputs to inputs propagator

For this section, outputs are given and we want to over-approximate with an interval (as small as possible) the set containing all the inputs that could generate those outputs. This is done by propagating all the constraints in the `real-time-loop` constraint until a fixpoint is reached. Indeed, since the outputs are fixed, propagating the constraints either reduce the input domains or do not change any domain. Since an input domain of a block can be an output domain of another block, we continue propagating the domains until no domain is modified. This procedure corresponds to the standard HC4 algorithm [66] from interval constraint programming.

## 4.5 Applications

We present three generic applications using our model for real-time programs that can be represented as Block-Diagrams.

**Verification** Program verification consists in checking properties of a given program written in a specific language. Block-Diagram programs are designed to run on a definite (possibly infinite) duration. Users may be interested in ensuring that no problem will occur during execution (especially if the software failure can impact damages). From a programmer point of view, one of the classic properties that can be checked is to ensure that some *strategic* or *critical* variables will stay into a specific interval. This problem is usually known as overflow checking. In our CP approach, fixing the input and then solving our model makes it possible to compute over-approximation for each stream/variable.

**Refactoring** Usually, a single semantics meaning can be implemented by many different syntactical writings. It is well known that the same result (even for a given algorithm) can be obtained by different implementations. Refactoring consists in restructuring an existing implementation without changing its external behavior. On Block-Diagrams, refactoring consists in removing or adding blocks or connections without changing the output values. For instance, an if-then-else condition which is always evaluated to "true" can be replaced by its "then" statement. This is a particular case of refactoring: removing *dead code*. In our CP approach, fixing inputs and outputs before solving enables removing blocks leading to empty over-approximations.

**Compilation Assistant** Block-Diagram is a high-level programming language designed to create Real-Time programs in an elegant and human readable way. As seen in the previous sections, such languages can manipulate delays. Note that these delays can be the result of a complex computation. This implies that the maximum delay may be unknown at compilation time. Thus it must be given at run time and at each-time step

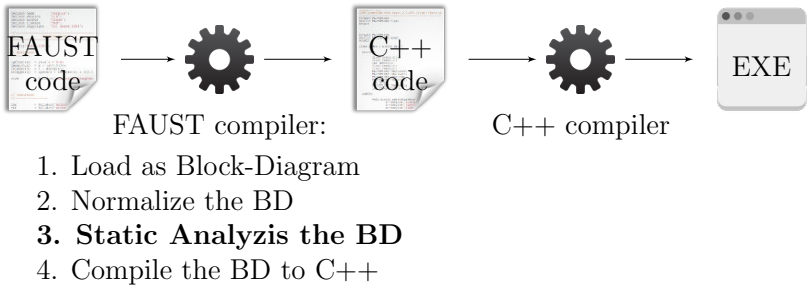



Figure 4.10: FAUST Compilation Scheme

(*i.e.*, the delay can change during execution). Hence, if the compiler is able to estimate the maximal delay, no value will be missing at execution time. With our CP model we can bound maximum delay for temporal blocks: such information can be given to the compiler in order to allocate appropriate arrays for saving delays.

## 4.6 Application to FAUST and Experiments

For the application section, we chose the Real-Time language FAUST and we focused on a verification problem. FAUST allows us to manipulate audio streams. To illustrate this section, we selected the VOLUME-CONTROLLER program (a real-world program) from the official set of examples as the running example. We first introduce the FAUST language, then the constraint programming model for verification problem, and finally we conclude with experiments over a set of real-world FAUST programs.

### 4.6.1 Model FAUST Programs

FAUST (Functional Audio Stream) has been designed for real-time signal processing and synthesis. Figure 4.10 presents the compilation scheme for creating FAUST applications. First, it needs a program, called the *source program*, written in the dedicated FAUST language (this language is not significant for our contribution and is similar to other languages designed for digital signal processing). See [68] for more details. Then, this source program must be compiled by the FAUST compiler. This produces a C++ program that can finally be compiled with a usual C++ compiler by targeting the desired device. This hatched process, allows a single FAUST program to run on phones, web browsers, concert devices, etc.

The goal of the FAUST compiler is to produce a C++ optimized code (*i.e.*, a code with good performances and well managed memory in order to run efficiently in real-time, even on small devices). The actual FAUST compiler already contains various technics from the compilation research field for tackling this objective. As shown on Figure 4.10, it operates in four steps:

| Block | Semantics | Constraint Model |
|---|---|---|
| $b = mem(a)$ | $\begin{cases} b(0) & = 0 \\ b(t) & = a(t-1), \text{ if } t > 0 \end{cases}$ | $b = [a \bigcup 0]$ |
| $c = delay(a, b)$ | $\begin{cases} c(t) & = 0, \text{ if } t < b(t) \\ c(t) & = a(t - b(t)), \text{ otherwise} \end{cases}$ | $c = [a \bigcup 0]$ |
| $c = prefix(a, b)$ | $\begin{cases} c(0) & = a(0) \\ c(t) & = b(t-1), \text{ if } t > 0 \end{cases}$ | $c = [a \bigcup b]$ |

Table 4.3: FAUST temporal blocks

- it loads the source program in an internal representation, easy to manipulate (*i.e.*, block-diagram)
- it rewrites this block-diagram to a normal form by syntactic analyzis (*e.g.*, simplifying redundant forms such as $x - x$ by 0)
- it performs a static analysis in order to compute approximations of the semantics of the program (*e.g.*, estimate the maximal size for a delay)
- and finally it produces the C++ program thanks to all the gathered information

In our experiments, we use the model proposed in the previous sections to *improve the static analysis* inside the FAUST compiler. To do so, we will consider the block-diagram just before the C++ code generation. Note that the normalization and the static analyzis made by the actual FAUST compiler helps working on expressions with few occurrences of the same variable: this is important for the constraint programming model since propagation over continuous variables performs poorly on variables occurring in many constraints [89] (*e.g.*, the stream "$s' = s - s$" equals to zero for all time while its constraint model over intervals "$S' = S - S$" is not equivalent to $[0; 0]$).

The FAUST language for writing source code has a formally well defined semantics in the Block-Diagram language [68] and is expressive thanks to: three temporal blocks (*prefix*, *mem*, and *delay*); common arithmetic functions (*e.g.*, addition, subtraction, ...); many C++ imported functions (*e.g.*, *sin*, *cos*, *exp*, ...); relational and conditional operators.[3] All these block operators admit an interval extension (as defined in Definition 4.3.3) with a natural translation to interval constraints. In particular, Table 4.3 presents the semantics and the models of the temporal blocks.

**Example 24.** Figure 4.11 is our running example in FAUST (the FAUST Volume Controller Program) while Figure 4.5 is its equivalent representation in block-diagram (note that this block-diagram is not in normal form since the constant expression $1 - 0.999$ has not been reduced to 0.001). When running this program with FAUST, the graphical interface presents a slider (*vslider* in the FAUST source code stands for "vertical slider")

[3] See `http://faust.grame.fr/index.php/documentation/references` for listing and description.

allowing to control the output volume (left sliding reduces the volume and right sliding increases the volume).

CP problems are formatted in three parts. The first one contains the variable declaration: it introduces the variables with their corresponding type (*e.g.*, integer, real-number). The second one precises a domain as an interval for each declared variable. The third one contains the constraints. Figure 4.13 depicts these parts for our running example using the Medium model presented in Section 4.3 and the optimized model presented in Section 4.4. We can read that: only Variables 10, 14, and 18 are over integers; Variables 8, 10, 12, 14, 16, 18 correspond to constants from the block-diagram; Variable 17 models the vslider with range/domain $[-70; 4]$; and Variable 2 stands for the input audio stream[4]. Note that the normalization performed by FAUST and used for our CP modelling replaced the constant expression "$1 - 0.999$" by the constant 0.001 (cf. Variable 12 in Figure 4.13b); replaced the expression "vslider / 20" by the expression "vslider $\times$ 0.05" (cf. in Figure 4.13b the constraint $[15] = \text{mul}(16, 17)$); and introduced identity operators (cf. identity constraints over the Variables 4 and 5 in Figure 4.13b). Even if the identity operators increase the size of the CP model, they will not affect the quality of the over-approximations (*i.e.*, identity propagation can be done without loss of precision). We discuss about this point and possible improvements in Section 4.7. The block-diagram contains one loop, and thus, it is not surprising to find out the corresponding `real-time-loop` constraint in the CP model (see Figure 4.13d).

### 4.6.2 Verifying FAUST Programs

We described how to model FAUST programs in CP. We now discuss about the CP solver. The solver has been implemented using IBEX. It is able to deal with two types of variables: *real-numbers* (*i.e.*, in practice approximated by floating-point numbers intervals) and integers (*i.e.*, *C++ int*). Table 4.4 presents the results for our benchmark programs. It is composed of some pathological DSP programs, and of real world programs from the FAUST standard library. They have been selected for their interest since they are basics for many bigger FAUST compositions. From left to right, columns of the table represent: the name of the FAUST program; the number of constraints followed by the number of variables in the medium model; the number of constraints followed by the number of variables in the optimized model; the number of `real-time-loop` constraints with the maximum number of constraints and the maximum number of arguments for the transfer function; the average time for compiling a FAUST program into the medium model; the average time for compiling the medium model into the optimized model; the average time for solving the optimized model; the over-approximation returned by the solver for the output stream associated with an indicator of reachability of the smallest

[4]In music, a numeric audio stream is a sequence of values between $-1$ and 1

```
declare author       "Grame";
declare license      "BSD";
declare copyright    "(c) GRAME 2006";

//-----------------------------------------------
// Title : Volume control in dB
// Remark: extracted from Faust examples
//-----------------------------------------------
import("music.lib");

smooth(c) = *(1-c) : +~*(c);
// vslider: - default value: 0
//          - range between: -70 and +4
//          - range with a step of: 0.1
gain      = vslider("[1]", 0, -70, +4, 0.1)
              : db2linear : smooth(0.999);
process   = *(gain);
```

Figure 4.11: FAUST Volume Controller Source Program

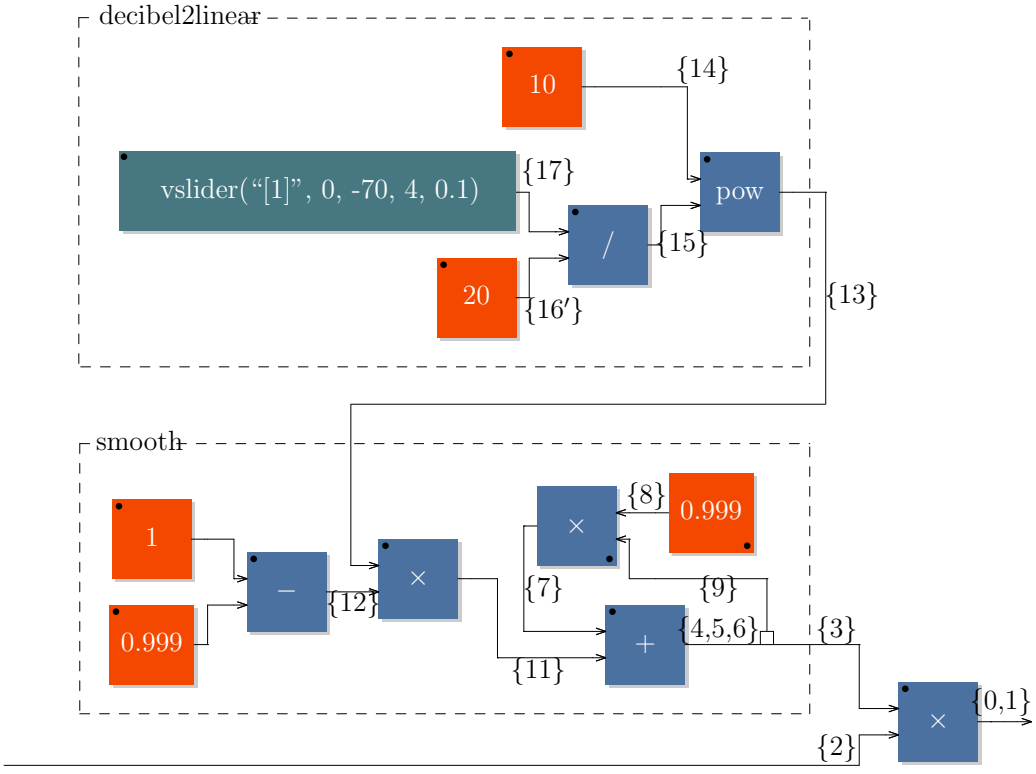



Figure 4.12: FAUST volume controller block-diagram before normalization. Edges are labeled with their corresponding variables in the CSP in Fig. 4.13b

```
0=Real
1=Real
2=Real
3=Real
4=Real
5=Real
6=Real
7=Real
8=Real
9=Real
10=Integer
11=Real
12=Real
13=Real
14=Integer
15=Real
16=Real
17=Real
18=Integer
```

(a) Variables Declaration

```
[0]=output_0(1)
[1]=mul(2,3)
[3]=delay(4,18)
[4]=id(5)
[5]=id(6)
[6]=add(7,11)
[7]=mul(8,9)
[9]=delay(4,10)
[11]=mul(12,13)
[13]=pow(14,15)
[15]=mul(16,17)
```

(b) Constraints for the medium model

```
0=[-oo;+oo]
1=[-oo;+oo]
2=[-1;1]
3=[-oo;+oo]
4=[-oo;+oo]
5=[-oo;+oo]
6=[-oo;+oo]
7=[-oo;+oo]
8=[0.999;0.999]
9=[-oo;+oo]
10=[1;1]
11=[-oo;+oo]
12=[0.001;0.001]
13=[-oo;+oo]
14=[10;10]
15=[-oo;+oo]
16=[0.05;0.05]
17=[-70;4]
18=[0;0]
```

(c) Variables Domain

```
[0]=output_0(1)
[1]=mul(2,3)
[3]=delay(4,18)
real-time-loop(
      [[4]=id(5),
       [5]=id(6),
       [6]=add(7,11),
       [7]=mul(8,9),
       [9]=delay(4,10)],
      [10,8,11],
      [4])
[11]=mul(12,13)
[13]=pow(14,15)
[15]=mul(16,17)
```

(d) Constraints for the optimized model

Figure 4.13: CSP for the VOLUME benchmark

over-approximation. This indicator is computed by hand and "√" stands for *verified over-approximation* by human while "?" stands for *unverified over-approximation* mainly due to the program complexity in term of number of blocks/streams. In order to get readable outputs, intervals are given in decimal format with a fixed precision of $10^{-2}$.

Among those programs, COUNTER is an incremental infinite loop starting at 0; NOISE generates a random noise (*i.e.*, sequence of random numbers) ; OSCILLATOR generates an oscillating sound wave, FREEVERB generates a reverb on the input stream, FIRST-ORDER-FILTER is well named and corresponds to a first-order filter. Note that benchmarks from COUNTER to FREEVERB presented in Table 4.4 are fundamental block-diagrams for building more complex programs by composition. As instances of aggregation, we propose a family of 6 benchmarks for additive synthesis [90] concataning from 5 to 1,000 of these fundamental block-diagrams (cf. ADD-SYNTH-X-OSCS benchmarks in Table 4.4). The whole benchmark description, with the detailed information (DSP, block-diagram, and models) for each program, can be found at `http://anicet.bart.free.fr/benchmarks/FAUST`.

Each call to the `real-time-loop` constraint propagator runs five times Algorithm 3 and returns the intersection of the computed over-approximations. Each call to Algorithm 3 is limited to 500 loop iterations/transfer function evaluations. The *selection heuristic* in Algorithm 3 does intelligent search by selecting a new bound for the moving/changing bound (*e.g.*, if the application of the transfer function does not change the lower bound of an interval, it will only select a new upper bound for the next evaluation). The selected precision for interval is $10^{-5}$. The solver has been launched 10 times for each benchmark and the averages of computation times and solutions on the 10 runs are presented in Table 4.4. Experimentation has been done on a 2.4 GHz Intel Core i5 processor with a memory limit set to 16 Go.

### 4.6.3 Results and Discussion

Results in Table 4.4 can be partitioned into three sets.

- COUNTER, PAPER-EXAMPLE, SINUS, NOISE, ALLPASS-FILTER, VOLUME, COMB-FILTER, ECHO, STEREO-ECHO, OSCILLATOR ADD-SYNTH-X-OSCS are benchmark programs for which the returned solution is the smallest over-approximation of the output stream, *i.e.*, the smallest interval containing all the possible values at any runtime execution. It is well known in abstract interpretation that first order filters, cannot generally be over-approximated efficiently using intervals. However, the FIRST-ORDER-FILTER benchmark is a special case (nevertheless a standard in FAUST) for which the floating-point interval abstraction is contracting.

- PINK-NOISE, CAPTURE, KARPLUS-STRONG, BAND-FILTER are benchmark programs

| Program name | #var | #cstrs medium model | #cstrs optim. model | real-time-loop # | real-time-loop max. cstrs | real-time-loop max. args | Time (in ms) comp. medium | Time (in ms) comp. optim. | Time (in ms) solve | Verification solver output | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COUNTER | 8 | 6 | 3 | 1 | 4 | 1 | 16 | 460 | 7 | [0; MAX] | √ |
| PAPER-EXAMPLE | 11 | 7 | 3 | 1 | 5 | 1 | 17 | 458 | 7 | [0; 1] | √ |
| SINUS | 9 | 7 | 4 | 1 | 4 | 1 | 15 | 462 | 7 | [-1; 1] | √ |
| FIRST-ORDER-FILTER | 15 | 10 | 6 | 1 | 5 | 1 | 35 | 473 | 9 | [-1; 1] | √ |
| NOISE | 16 | 10 | 6 | 1 | 5 | 1 | 16 | 454 | 8 | [-1; 1] | √ |
| ALLPASS-FILTER | 16 | 11 | 6 | 1 | 6 | 1 | 18 | 470 | 9 | [-3; 3] | √ |
| VOLUME | 19 | 11 | 7 | 1 | 5 | 1 | 25 | 473 | 7 | [-1.58; 1.58] | √ |
| COMB-FILTER | 20 | 15 | 5 | 1 | 11 | 1 | 18 | 462 | 7 | [-oo; +oo] | √ |
| ECHO | 29 | 19 | 15 | 1 | 5 | 1 | 27 | 482 | 8 | [-oo; +oo] | √ |
| STEREO-ECHO | 37 | 26 | 18 | 2 | 5 | 1 | 28 | 495 | 12 | [-oo; +oo] | √ |
| PINK-NOISE | 40 | 28 | 15 | 2 | 10 | 1 | 27 | 493 | 7 | [-oo; +oo] | √ |
| CAPTURE | 45 | 34 | 21 | 3 | 6 | 1 | 27 | 488 | 14 | [-oo; +oo] | √ |
| KARPLUS-STRONG | 49 | 35 | 18 | 3 | 8 | 1 | 30 | 484 | 9 | [-oo; +oo] | √ |
| OSCILLATOR | 49 | 35 | 23 | 3 | 6 | 1 | 30 | 497 | 11 | [-1; 1] | √ |
| BAND-FILTER | 55 | 42 | 34 | 1 | 9 | 1 | 38 | 546 | 11 | [-oo; +oo] | √ |
| LOWBOOST | 59 | 46 | 38 | 1 | 9 | 1 | 33 | 508 | 10 | [-oo; +oo] | ? |
| PITCH-SHIFTER | 60 | 50 | 46 | 1 | 5 | 1 | 32 | 510 | 8 | [-59902;59902] | ? |
| SMOOTH-DELAY | 100 | 85 | 25 | 3 | 43 | 4 | 40 | 789 | 17 | [-oo; +oo] | ? |
| MIXER | 356 | 310 | 234 | 19 | 5 | 1 | 65 | 824 | 49 | [-20.01;20.01] | ? |
| FREEVERB | 371 | 335 | 103 | 24 | 13 | 1 | 69 | 994 | 41 | [-oo; +oo] | ? |
| HARPE | 407 | 348 | 197 | 24 | 8 | 1 | 76 | 935 | 52 | [-oo; +oo] | ? |
| ADD-SYNTH-5-OSCS | 106 | 85 | 54 | 7 | 6 | 1 | 84 | 605 | 15 | [-1; 1] | √ |
| ADD-SYNTH-10-OSCS | 181 | 150 | 94 | 12 | 6 | 1 | 110 | 689 | 17 | [-1; 1] | √ |
| ADD-SYNTH-50-OSCS | 780 | 670 | 414 | 52 | 6 | 1 | 244 | 1,108 | 86 | [-1; 1] | √ |
| ADD-SYNTH-100-OSCS | 1,530 | 1,320 | 814 | 102 | 6 | 1 | 609 | 1.6s | 314 | [-1; 1] | √ |
| ADD-SYNTH-250-OSCS | 3,780 | 3,270 | 2,014 | 252 | 6 | 1 | 2.5s | 4.5s | 1.6s | [-1; 1] | √ |
| ADD-SYNTH-500-OSCS | 7,530 | 6,520 | 4,014 | 502 | 6 | 1 | 12.5s | 17.3s | 10.1s | [-1; 1] | √ |
| ADD-SYNTH-750-OSCS | 11,280 | 9,770 | 6,014 | 752 | 6 | 1 | 39.8s | 1'18s | 48.8s | [-1; 1] | √ |
| ADD-SYNTH-1000-OSCS | 15,030 | 13,020 | 8,014 | 1,002 | 6 | 1 | 1'25s | 2'43s | 2'34s | [-1; 1] | √ |

Table 4.4: Experimental results on a benchmark of FAUST programs

for which the returned solution is the smallest over-approximation of the output stream using interval analyses. Indeed, the analysis/propagation is made block by block/constraint by constraint and some patterns cannot give small over-approximation without knowing local semantics such as *e.g.*, $x - \texttt{floor}(x)$ corresponds to the *decimal part of* $x$.

- for the other programs (see lines in Table 4.4 containing the "?" symbol), the returned solution may not be the smallest over-approximation of the output stream but we were not able to prove it by hand.

In order to be included into the FAUST compiler, the verification must be executed in a very short time (more or less a second). For our experiments, the solver performs well on that matter: even in rather complex programs (such as FREEVERB or HARPE) it is able to answer quickly. For most of the programs, the longest task is to compile the medium model into the optimized model. This is due to the use of an external library to represent graph data structures and compute strongly connected components. However it seems to have

good scalability: Table 4.4 shows that even when the size of the benchmark program is multiplied by more than 20, the execution time is only multiplied by 2. Finally, according to the over-approximations computed with our method, a FAUST user expert confirmed the existence of saturation in 3 programs: VOLUME, PITCH-SHIFTER, and MIXER. The saturation came from the fundamental VOLUME FAUST program, which contained an incorrectly set constant (*i.e.*, a vslider ranging from $[-70; 4]$ instead of $[-70; 2]$). Due to the execution time of our method and the quality of the returned solutions, the FAUST developers shown a big interest for integrating our contribution in a future version of the official FAUST compiler. Nevertheless, note that our ADD-SYNTH-X-OSCS benchmarks ranging from 5 to 1,000 oscillators illustrates an exponentiel tendance in compiling and solving time, compared to the block-diagram size.

### 4.6.4 Related works

The research on Constraint Programming and Verification has always been rich, and gained a great interest in the past decade. Constraint Programming has been applied to verification for test generation (see [57] for an overview), constraint-based model-checking ([73]), control-flow graph analysis [74], or even worst-execution time estimations ([75]). More recently, detailed approaches have been presented by [76] or [77] to carefully analyze floating-points conditions with continuous constraint methods.

Our contribution mixes CP and Abstract Interpretation. It has been known for a long time that both domains shared a lot of ideas, since for instance [59] which expresses the constraint consistency as chaotic iterations. A key remark is the following: Abstract Interpretation is about over-approximating the traces of a program, and Constraint Programming uses propagation to over-approximate a solution set. It is worth mentioning that one of the over-approximation algorithms used in Abstract Interpretation, the bottom-up top-down algorithm for the interval abstraction [54, 78], is the same as the HC4 constraint propagator [66]. Recent works explored this links in both ways, either to refine CP techniques [60], or to improve the Abstract Interpretation analysis [81].

As a close work in the Constraint Programming community, GATeL [82] is a software based on logical constraint programming verifying real-time programs. This tool first translates a Lustre program (representable as a block-diagram) and the specification of its environment in an equivalent Prolog representation, *i.e.*, in a Constraint Logic Program (CLP). Then, it adds the user defined test objective in the CLP and solves it, computing a test input satisfying the objective for the given Lustre program. This work already gathers the CP and the verification of real-time programs communities. However, while Gatel performs test cases generation for real-time programs we are interesting in finding precise over-approximations.

As a close work in the Abstract Interpretation community, ReaVer[5] is a state-of-the-art software for safety verification of data-flow languages, like Lustre, Lucid Synchrone or Zelus (all are close to FAUST), providing time-unbounded analysis based on abstract interpretation techniques. It features partitioning techniques and several analysis methods [91] (*e.g.*, Kleene iteration based methods with increasing and descending iterations, abstract acceleration, max-strategy iteration, and relational abstractions; logico-numerical product and power domains with convex polyhedra, octagons, intervals, and template polyhedra). Considering our problem of over-approximating stream in block-diagrams, while a solver like ReaVer embarks many technics from Abstract Interpretation to answer this problem, in our approach we focus on how a slightly modified Constraint Programming solver can be turned into a verification tool with good performances (*i.e.*, in computation time and in over-approximations qualities). Experiments in the previous section show that our `real-time-loop` constraint together with the proposed propagators achieve these objectives. However, it is clear that this approach is not competitive when the interval abstract domain cannot tightly fit the concrete domain (*i.e.*, in these cases, polyhedra, octagons, or an other domains may provide better over-approximations). In some cases the interval $[-\infty, +\infty]$ is returned as over-approximation of the output streams, which is indeed the smallest over-approximation of the output streams in the interval abstract domain while $[-1, 1]$ is a valid over-approximation of the concrete output stream.

## 4.7 Conclusion and Perspectives

**Conclusion** We proposed a constraint model using a global constraint for over-approximation of real-time streams represented with block-diagrams. The experiments show that our approach can reach very good, nearly always optimal, over-approximations in a short running time. Our method has been taken in consideration for a future implementation into the FAUST compiler.

In addition, we showed that constraint programming can handle block-diagram analyses in an elegant and natural way. The concept of digital signal processing is not proper to FAUST nor to audio processing. Indeed, it also appears in a lot of applications receiving and processing digital signals: modems, multimedia devices, GPS, video processing; which empower this model. Thus, this gives good perspectives for this work.

**Perspectives** The results of our experiments are fast and of good quality. However, we would like to point out some possible improvements. A common way to improve performances is to consider pre-processing. This consists in taking advantage of some knowledge about the semantics of the problem in order to find faster a solution. According to the application (*e.g.*, verification, refactoring) it could be interesting to propagate

[5] https://www.cs.ox.ac.uk/people/peter.schrammel/reaver/

variables with respect to a global order. For instance for verification, it will be faster to propagate from inputs to outputs instead of a totally arbitrary order. Algorithm 3 for stable interval search applies many times the transfer function of the loop. Thus, reducing the number of blocks per transfer function would have two impacts: decreasing the time needed by the solver, and decreasing the number of variable multiple occurrences. Factoring sets of blocks with specific semantics would lead to better models from which faster and better over-approximation would be computed. For example, removing identity constraints, factoring sub block-diagram with specific meaning such as filter would lead to better models.

Chapter 5

# Verifying Parametric Interval Markov Chains with Constraints

## Contents

This chapter treats model checking of qualitative and quantitative properties over abstractions of Markov chains. In particular, we show in the qualitative context how constraint modellings produce better models in terms of size and resolution time. We also present a formal theorem allowing to produce a first practical approach for verifying some quantitative properties on the considered Markov chain abstractions. Finally, we propose an implementation of our modellings and discuss the results. This chapter is self-contained including introduction, motivation, background, state of the art, and contributions.

## 5.1 Introduction

Discrete time Markov chains (MCs for short) are a standard probabilistic modelling[1] formalism that has been extensively used in the litterature to reason about software [92] and real-life systems [93]. However, when modelling real-life systems, the exact value of transition probabilities may not be known precisely. Several formalisms abstracting MCs have therefore been developed. Parametric Markov chains [94] (pMCs for short) extend MCs by allowing parameters to appear in transition probabilities. In this formalism, parameters are variables and transition probabilities may be expressed as polynomials over these variables. A given pMC therefore represents a potentially infinite set of MCs, obtained by replacing each parameter by a given value. pMCs are particularly useful to represent systems where dependencies between transition probabilities are required. Indeed, a given parameter may appear in several distinct transition probabilities, therefore requiring that the same value is given to all its occurences. Interval Markov chains [95] (IMCs for short) extend MCs by allowing precise transition probabilities to be replaced by intervals, but cannot represent dependencies between distinct transitions. IMCs have mainly been studied with three distinct semantics interpretations. Under the *once-and-for-all* semantics, a given IMC represents a potentially infinite number of MCs where transition probabilities are chosen inside the specified intervals while keeping the same underlying graph structure. The *interval-Markov-decision-process* semantics (IMDP for short), such as presented in [96, 97], does not require MCs to preserve the underlying graph structure of the original IMC but instead allows an "unfolding" of the original graph structure: new probability values inside the intervals can be chosen each time a state is visited. Finally, the *at-every-step* semantics, which was the original semantics given to IMCs in [95], does not preserve the underlying graph structure too while allowing to "aggregate" and "split" states of the original IMC in the manner of probabilistic simulation.

Model-checking algorithms and tools have been developed in the context of pMCs [98, 99, 100] and IMCs with the once-and-for-all and the IMDP semantics [101, 102]. State of the art tools [98] for pMC verification compute a rational function on the parameters

[1] In this chapter we use modelling with the verification meaning and we call encoding a CSP modelling.

that characterizes the probability of satisfying a given property, and then use external tools such as SMT solving [98] for computing the satisfying parameter values. For these methods to be viable in practice, the allowed number of parameters is quite limited. On the other hand, the model-checking procedure for IMCs presented in [102] is adapted from machine learning and builds successive refinements of the original IMCs that optimize the probability of satisfying the given property. This algorithm converges, but not necessarilly to a global optimum. It is worth noticing that existing model checking procedures for pMCs and IMCs strongly rely on their underlying graph structure (*i.e.*, respect the once-and-for-all semantics). However, in [96] the authors perform model checking of $\omega$-PCTL formulas on IMCs w.r.t. the IMDP semantics and they show that model checking of LTL formulas can be solved for the IMDP semantics by reduction to the model checking problem of $\omega$-PCTL on IMCs with the IMDP semantics. For all that, to the best of our knowledge, no solutions for model-checking IMCs with the at-every-step semantics have been proposed yet.

In this thesis chapter, we focus on Parametric interval Markov chains [103] (pIMCs for short), that generalize both IMCs and pMCs by allowing parameters to appear in the endpoints of the intervals specifying transition probabilities, and we provide four main contributions.

First, we formally compare abstraction formalisms for MCs in terms of succinctness: we show in particular that pIMCs are *strictly more succinct* than both pMCs and IMCs when equipped with the right semantics. In other words, everything that can be expressed using pMCs or IMCs can also be expressed using pIMCs while the reverse does not hold.

Second, we prove that the once-and-for-all, the IMDP, and the at-every-step semantics are equivalent w.r.t. reachability properties, both in the IMC and in the pIMC settings. Notably, this result gives theoretical backing to the generalization of existing works on the verification of IMCs to the at-every-step semantics.

Third, we study the parametric verification of fundamental properties at the pIMC level: consistency, qualitative reachability, and quantitative reachability. Given the expressivity of the pIMC formalism, the risk of producing a pIMC specification that is incoherent and therefore does not model any concrete MC is high. We therefore propose constraint encodings for deciding whether a given pIMC is consistent and, if so, synthesizing parameter values ensuring consistency. We then extend these encodings to qualitative reachability, *i.e.*, ensuring that given state labels are reachable in *all* (resp. *none*) of the MCs modelled by a given pIMC. Finally, we focus on the quantitative reachability problem, *i.e.*, synthesizing parameter values such that the probability of reaching given state labels satisfies fixed bounds in *at least one* (resp. *all*) MCs modelled by a given pIMC. While consistency and qualitative reachability for pIMCs have already been studied in [103], the constraint encodings we propose are significantly smaller (linear instead of exponential). To the best of our knowledge, our results provide the first solution to the

quantitative reachability problem for pIMCs. Our last contribution is the implementation of all our verification algorithms in a prototype tool that generates the required constraint encodings and can be plugged to any SMT solver for their resolution.

## 5.2 Background

In this section we introduce notions and notations that will be used throughout this chapter. Given a finite set of variables $X = \{x_1, \ldots, x_k\}$, we write $D_x$ for the domain of the variable $x \in X$ and $D_X$ for the set of domains associated to the variables in $X$. A valuation $v$ over $X$ is a set $v = \{(x, d) | x \in X, d \in D_x\}$ of elementary valuations $(x, d)$ where for each $x \in X$ there exists a unique pair of the form $(x, d)$ in $v$. When clear from the context, we write $v(x) = d$ for the value given to variable $x$ according to valuation $v$. A rational function $f$ over $X$ is a division of two (multivariate) polynomials $g_1$ and $g_2$ over $X$ with rational coefficients, *i.e.*, $f = g_1/g_2$. We write $\mathbb{Q}$ for the set of rational numbers and $\mathbb{Q}_X$ for the set of rational functions over $X$. The evaluation $v(g)$ of a polynomial $g$ under the valuation $v$ replaces each variable $x \in X$ by its value $v(x)$.

An *atomic constraint* over $X$ is a Boolean expression of the form $f(X) \bowtie g(X)$, with $\bowtie \in \{\leq, \geq, <, >, =\}$ and $f$ and $g$ two functions over variables in $X$. An atomic constraint is *linear* if the functions $f$ and $g$ are linear. A *constraint* over $X$ is a Boolean combination of atomic constraints over $X$.

Given a finite set of states $S$, we write $\mathsf{Dist}(S)$ for the set of probability distributions over $S$, *i.e.*, the set of functions $\mu : S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$. We write $\mathbb{I}$ for the set containing all the interval subsets of $[0, 1]$. In the following, we consider a universal set of symbols $A$ that we use for labelling the states of our structures. We call these symbols *atomic propositions*. We will use Latin alphabet in state context and Greek alphabet in atomic proposition context.

**Constraints.** Constraints are first order logic predicates used for modelling and solving combinatorial problems [104]. A problem is described with a list of variables, each in a given domain of possible values, together with a list of constraints over these variables. Such problems are then sent to solvers which decide whether the problem is satisfiable, *i.e.*, if there exists a valuation of the variables satisfying all the constraints, and in this case compute a solution. Recall that checking satisfiability of constraint problems is difficult in general (cf. Chapter 2).

Formally, a Constraint Satisfaction Problem (CSP) is a tuple $\Omega = (X, D, C)$ where $X$ is a finite set of variables, $D = D_X$ is the set of all the domains associated to the variables from $X$, and $C$ is a set of constraints over $X$. We say that a valuation over $X$ satisfies $\Omega$ if and only if it satisfies all the constraints in $C$. We write $v(C)$ for the satisfaction result of the valuation of the constraints $C$ according to $v$ (*i.e.*, true or false). In the following we call CSP *encoding* a scheme for formulating a given problem into a CSP. The size of

a CSP corresponds to the number of variables and atomic constraints appearing in the problem. Note that, in constraint programming, having less variables or less constraints during the encoding does not necessarily imply faster solving time of the problems.

**Discrete Time Markov Chains.** A Discrete Time Markov Chain (DTMC or MC for short) is a tuple $\mathcal{M} = (S,\ s_0,\ p,\ V)$, where $S$ is a finite set of states containing the initial state $s_0$, $V : S \to 2^A$ is a labelling function, and $p : S \to \mathsf{Dist}(S)$ is a probabilistic transition function. We write $\mathtt{MC}$ for the set containing all the discrete time Markov chains.

A Markov Chain can be represented as a directed graph where the nodes correspond to the states of the MC and the edges are labelled with the probabilities given by the transition function of the MC. In this representation, a missing transition between two states represents a transition probability of zero. As usual, given an MC $\mathcal{M}$, we call a *path* of $\mathcal{M}$ a sequence of states obtained from executing $\mathcal{M}$, *i.e.*, a sequence $\omega = s_1, s_2, \ldots$ such that the probability of taking the transition from $s_i$ to $s_{i+1}$ is strictly positive, $p(s_i)(s_i + 1) > 0$, for all $i$. A path $\omega$ is finite iff it belongs to $S^*$, *i.e.*, it represents a finite sequence of transitions from $\mathcal{M}$.

**Example 25.** Figure 5.1 illustrates the Markov chain $\mathcal{M}_1 = (S, s_0, p, V) \in \mathtt{MC}$ where the set of states $S$ is given by $\{s_0, s_1, s_2, s_3, s_4\}$, the atomic proposition are restricted to $\{\alpha, \beta\}$, the initial state is $s_0$, and the labelling function $V$ corresponds to $\{(s_0, \emptyset)$, $(s_1, \{\alpha\})$, $(s_2, \{\beta\})$, $(s_3, \{\alpha, \beta\})$, $(s_4, \alpha)\}$. The sequences of states $(s_0, s_1, s_2)$, $(s_0, s_2)$, and $(s_0, s_2, s_2, s_2)$, are three (finite) paths from the initial state $s_0$ to the state $s_2$.

**Reachability.** A Markov chain $\mathcal{M}$ defines a unique probability measure $\mathbb{P}^{\mathcal{M}}$ over the paths from $\mathcal{M}$. According to this measure, the probability of a finite path $\omega = s_0, s_1, \ldots, s_n$ in $\mathcal{M}$ is the product of the probabilities of the transitions executed along this path, *i.e.*, $\mathbb{P}^{\mathcal{M}}(\omega) = p(s_0)(s_1) \cdot p(s_1)(s_2) \cdot \ldots \cdot p(s_{n-1})(s_n)$. This measure naturally extends to infinite paths (see [105]) and to sequences of states over $S$ that are not paths of $\mathcal{M}$ by giving them a zero probability.

Given an MC $\mathcal{M}$, the overall probability of reaching a given state $s$ from the initial state $s_0$ is called the *reachability probability* and written $\mathbb{P}^{\mathcal{M}}_{s_0}(\Diamond s)$ or $\mathbb{P}^{\mathcal{M}}(\Diamond s)$ when clear from the context. This probability is computed as the sum of the probabilities of all finite paths starting in the initial state and reaching this state for the first time. Formally, let $\mathsf{reach}_{s_0}(s) = \{\omega \in S^* \mid \omega = s_0, \ldots s_n \text{ with } s_n = s \text{ and } s_i \neq s\ \forall 0 \leq i < n\}$ be the set of such paths. We then define $\mathbb{P}^{\mathcal{M}}(\Diamond s) = \sum_{\omega \in \mathsf{reach}_{s_0}(s)} \mathbb{P}^{\mathcal{M}}(\omega)$ if $s \neq s_0$ and 1 otherwise. This notation naturally extends to the reachability probability of a state $s$ from a state $t$ that is not $s_0$, written $\mathbb{P}^{\mathcal{M}}_t(\Diamond s)$ and to the probability of reaching a label $\alpha \subseteq A$ written $\mathbb{P}^{\mathcal{M}}_{s_0}(\Diamond \alpha)$. In the following, we say that a state $s$ (resp. a label $\alpha \subseteq A$) is reachable in $\mathcal{M}$ iff the reachability probability of this state (resp. label) from the initial state is strictly positive.

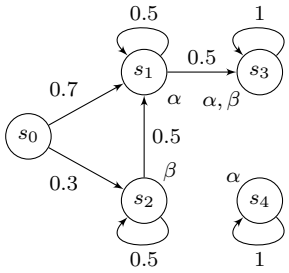



Figure 5.1: MC $\mathcal{M}_1$

**Example 26** (Example 25 continued)**.** In Figure 5.1 the probability of the path ($s_0$, $s_2$, $s_1$, $s_1$, $s_3$) is $0.3{\cdot}0.5{\cdot}0.5{\cdot}0.5 = 0.0375$ and the probability of reaching the state $s_1$ from $s_0$ is $\mathbb{P}_{s_0}^{\mathcal{M}_1}(\Diamond s_1) = p(s_0)(s_1) + \Sigma_{i=0}^{+\infty} p(s_0)(s_2){\cdot}p(s_2)(s_2)^i{\cdot}p(s_2)(s_1) = p(s_0)(s_1) + p(s_0)(s_2){\cdot}p(s_2)(s_1){\cdot}(1/(1-p(s_2)(s_2))) = 1$. Furthermore, the probability of reaching $\beta$ corresponds to the probability of reaching the state $s_2$, which is 0.3 here.

## 5.3 Markov Chain Abstractions

Modelling an application as a Markov Chain requires knowing the exact probability for each possible transition of the system. However, this can be difficult to compute or to measure in the case of a real-life application (*e.g.*, because of precision errors or limited knowledge). In this section, we start with a generic definition of Markov chain abstraction models. Then we recall three abstraction models from the literature, respectively pMC, IMC, and pIMC, and finally we present a comparison of these existing models in terms of succinctness.

**Definition 5.3.1** (Markov chain Abstraction Model)**.** *A Markov chain abstraction model (an abstraction model for short) is a pair* $(\mathtt{L}, \models)$ *where* $\mathtt{L}$ *is a nonempty set and* $\models$ *is a relation between* $\mathtt{MC}$ *and* $\mathtt{L}$*. Let* $\mathcal{P}$ *be in* $\mathtt{L}$ *and* $\mathcal{M}$ *be in* $\mathtt{MC}$ *we say that* $\mathcal{M}$ *implements* $\mathcal{P}$ *iff* $(\mathcal{M}, \mathcal{P})$ *belongs to* $\models$ *(*i.e.*,* $\mathcal{M} \models \mathcal{P}$*). When the context is clear, we do not mention the satisfaction relation* $\models$ *and only use* $\mathtt{L}$ *to refer to the abstraction model* $(\mathtt{L}, \models)$*.*

A *Markov chain Abstraction Model* is a specification theory for MCs. It consists in a set of abstract objects, called *specifications*, each of which representing a (potentially infinite) set of MCs – *implementations* – together with a satisfaction relation defining the link between implementations and specifications. As an example, consider the powerset of $\mathtt{MC}$ (*i.e.*, the set containing all the possible sets of Markov chains). Clearly, $(2^{\mathtt{MC}}, \in)$ is a Markov chain abstraction model, which we call the *canonical abstraction model*. This abstraction model has the advantage of representing all the possible sets of Markov chains but it also has the disadvantage that some Markov chain abstractions are only representable by an infinite extension representation. Indeed, recall that there exists

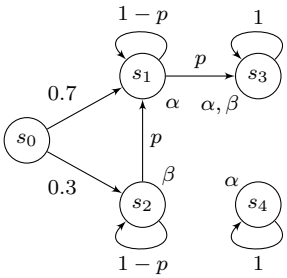



Figure 5.2: pMC $\mathcal{I}'$

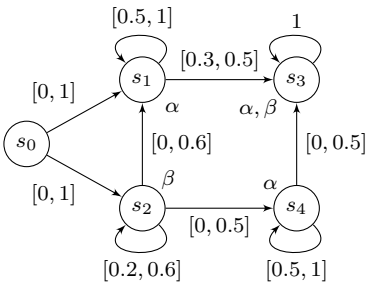



Figure 5.3: IMC $\mathcal{I}$

subsets of $[0, 1] \subseteq \mathbb{R}$ which cannot be represented in a finite space (e.g., the Cantor set [106]). We now present existing MC abstraction models from the literature.

### 5.3.1 Existing MC Abstraction Models

**Parametric Markov Chain** is an MC abstraction model from [94] where a transition can be annotated by a rational function over *parameters*. We write `pMC` for the set containing all the parametric Markov chains.

**Definition 5.3.2** (Parametric Markov Chain)**.** *A Parametric Markov Chain (*pMC *for short) is a tuple $\mathcal{I} = (S, s_0, P, V, Y)$ where $S$, $s_0$, and $V$ are defined as for* MC*s, $Y$ is a set of variables (parameters), and $P : S \times S \to \mathbb{Q}_Y$ associates with each potential transition a parameterized probability.*

Let $\mathcal{M} = (S, s_0, p, V)$ be an MC and $\mathcal{I} = (S', s'_0, P, V', Y)$ be a pMC. The satisfaction relation $\models_{\mathtt{p}}$ between `MC` and `pMC` is defined by $\mathcal{M} \models_{\mathtt{p}} \mathcal{I}$ iff $S = S'$, $s_0 = s'_0$, $V = V'$, and there exists a valuation $v$ of $Y$ such that $p(s)(s')$ equals $v(P(s, s'))$ for all $s, s'$ in $S$.

**Example 27.** Figure 5.2 shows a pMC $\mathcal{I}' = (S, s_0, P, V, Y)$ where $S$, $s_0$, and $V$ are identical to those of the MC $\mathcal{M}$ from Figure 5.1, the set $Y$ contains only one variable $p$, and the parametric transitions in $P$ are given by the edge labelling (*e.g.*, $P(s_0, s_1) = 0.7$, $P(s_1, s_3) = p$, and $P(s_2, s_2) = 1 - p$). Note that the pMC $\mathcal{I}'$ is a specification containing the MC $\mathcal{M}$ from Figure 5.1.

**Interval Markov Chains** extend MCs by allowing to label transitions with intervals of possible probabilities instead of precise probabilities. We write `IMC` for the set containing all the interval Markov chains.

**Definition 5.3.3** (Interval Markov Chain [95])**.** *An Interval Markov Chain (*IMC *for short) is a tuple $\mathcal{I} = (S, s_0, P, V)$, where $S$, $s_0$, and $V$ are defined as for* MC*s, and $P : S \times S \to \mathbb{I}$ associates with each potential transition an interval of probabilities.*

**Example 28.** Figure 5.3 illustrates IMC $\mathcal{I} = (S, s_0, P, V)$ where $S$, $s_0$, and $V$ are similar to the MC given in Figure 5.1. By observing the edge labelling we see that $P(s_0, s_1) =$

$[0,1]$, $P(s_1,s_1) = [0.5,1]$, and $P(s_3,s_3) = [1,1]$. On the other hand, the intervals of probability for missing transitions are reduced to $[0,0]$, e.g., $P(s_0,s_0) = [0,0]$, $P(s_0,s_3) = [0,0]$, $P(s_1,s_4) = [0,0]$.

In the litterature, IMCs have been mainly used with three distinct semantics: *at-every-step*, *interval-Markov-decision-process* and *once-and-for-all*. All these semantics are associated with distinct satisfaction relations which we now introduce.

The *once-and-for-all* IMC semantics [98, 107, 108] is alike to the semantics for pMC, as introduced above. The associated satisfaction relation $\models^o_{\mathtt{I}}$ is defined as follows: An MC $\mathcal{M} = (T, t_0, p, V^M)$ satisfies an IMC $\mathcal{I} = (S, s_0, P, V^I)$ iff $(T, t_0, V^M) = (S, s_0, V^I)$ and for all reachable state $s$ and all state $s' \in S$, $p(s)(s') \in P(s,s')$. In this sense, we say that MC implementations using the once-and-for-all semantics need to have the same structure as the IMC specification.

Next, the *interval-Markov-decision-process* IMC semantics (IMDP for short) [96, 97] operates as an "unfolding" of the original IMC by picking each time a state is visited a possibly new probability distribution which respects the intervals of probabilities. Thus, this semantics allows to produce MCs satisfying IMCs with a different structure. Formally, the associated satisfaction relation $\models^d_{\mathtt{I}}$ is defined as follows: An MC $\mathcal{M} = (T, t_0, p, V^M)$ satisfies an IMC $\mathcal{I} = (S, s_0, P, V^I)$ iff there exists a mapping $\pi$ from $T$ to $S$ s.t. $\pi(t_0) = s_0$, $V^I(\pi(t)) = V^M(t)$ for all state $t \in T$, $p(t)(t') \in P(\pi(t), \pi(t'))$ for all pair of states $t, t'$ in $T$, and for all state $t \in T$ and all state $s \in S$ there exists at most one state $t' \in \mathtt{Succ}(t)$ such that $\pi(t') = s$. Thus, we have that $\models^d_{\mathtt{I}}$ is more general than $\models^o_{\mathtt{I}}$ (*i.e.*, whenever $\mathcal{M} \models^o_{\mathtt{I}} \mathcal{I}$ we also have $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$). Note that in [96, 97] the authors allows the Markov chains satisfying the IMCs w.r.t. $\models^d_{\mathtt{I}}$ to have an infinite state space. In this work we consider Markov chains with a finite state space.

Finally, the *at-every-step* IMC semantics, first introduced in [95], operates as a simulation relation based on the transition probabilities and state labels, and therefore allows MC implementations to have a different structure than the IMC specification. Compared to the previous semantics, in addition to the unfoldings this one allows to "aggregate" and "split" states from the original IMC. Formally, the associated satisfaction relation $\models^a_{\mathtt{I}}$ is defined as follows: An MC $\mathcal{M} = (T, t_0, p, V^M)$ satisfies an IMC $\mathcal{I} = (S, s_0, P, V^I)$ iff there exists a relation $\mathcal{R} \subseteq T \times S$ such that $(t_0, s) \in \mathcal{R}$ and whenever $(t,s) \in \mathcal{R}$, we have 1. the labels of $s$ and $t$ correspond: $V^M(t) = V^I(s)$, 2. there exists a correspondence function $\delta : T \to (S \to [0,1])$ s.t. a) $\forall t' \in T$ if $p(t)(t') > 0$ then $\delta(t')$ is a distribution on $S$ b) $\forall s' \in S : (\Sigma_{t' \in T} p(t)(t') \cdot \delta(t')(s')) \in P(s,s')$, and c) $\forall (t',s') \in T \times S$, if $\delta(t')(s') > 0$, then $(t',s') \in \mathcal{R}$.

Example 29 illustrates the three IMC semantics and Proposition 3 compares them. We say that an IMC semantics $\models_1$ is more general than another IMC semantics $\models_2$ iff for all IMC $\mathcal{I}$ and for all MC $\mathcal{M}$ if $\mathcal{M} \models_2 \mathcal{I}$ then $\mathcal{M} \models_1 \mathcal{I}$. Also, $\models_1$ is strictly more general than $\models_2$ iff $\models_1$ is more general than $\models_2$ and $\models_2$ is not more general than $\models_1$.

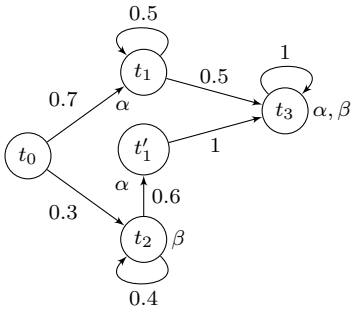



Figure 5.4: MC $\mathcal{M}_2$ satisfying the IMC $\mathcal{I}$ from Figure 5.3 w.r.t. $\models^d_{\mathtt{I}}$

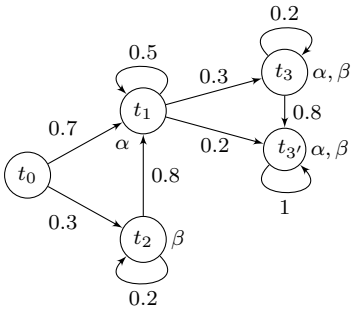



Figure 5.5: MC $\mathcal{M}_3$ satisfying the IMC $\mathcal{I}$ from Figure 5.3 w.r.t. $\models^a_{\mathtt{I}}$

**Example 29** (Example 28 continued)**.** Consider the IMC $\mathcal{I}$ from Figure 5.3, the MC $\mathcal{M}_1$ from Figure 5.1, the MC $\mathcal{M}_2$ from Figure 5.4 and the MC $\mathcal{M}_3$ from Figure 5.5. We have that $\mathcal{M}_1$ satisfies $\mathcal{I}$ w.r.t. $\models^o_{\mathtt{I}}$ and we say that $\mathcal{M}_1$ has the same structure than $\mathcal{I}$. By Proposition 3 we have by implication that $\mathcal{M}_1$ satisfies $\mathcal{I}$ w.r.t. $\models^d_{\mathtt{I}}$ and $\models^a_{\mathtt{I}}$. Regarding $\mathcal{M}_2$, we have that $\mathcal{M}_2$ satisfies $\mathcal{I}$ w.r.t. $\models^d_{\mathtt{I}}$. Note that two probability distributions have been chosen for the state $s_1$ from $\mathcal{I}$. This produces two states $t_1$ and $t'_1$ in $\mathcal{M}_2$ and changes the structure of the graph. Thus, $\mathcal{M}_2 \not\models^o_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}_2 \models^a_{\mathtt{I}} \mathcal{I}$. Finally, in the MC $\mathcal{M}_3$ with state space $T$ the state $s_3$ from $\mathcal{I}$ has been "split" into two states $t_3$ and $t_{3'}$ and the state $t_1$ "aggregates" the states $s_1$ and $s_4$ from $\mathcal{I}$. The relation $\mathcal{R} \subseteq T \times S$ containing the pairs $(t_0, s_0)$, $(t_1, s_1)$, $(t_1, s_4)$, $(t_2, s_2)$, $(t_3, s_3)$, and $(t_{3'}, s_3)$ is a satisfaction relation between $\mathcal{M}_2$ and $\mathcal{I}$ such as defined by $\models^a_{\mathtt{I}}$. Thus, $\mathcal{M}_3 \models^a_{\mathtt{I}} \mathcal{I}$. On the other hand, $\mathcal{M}_3 \not\models^d_{\mathtt{I}} \mathcal{I}$ since there exist probabilities on transitions that cannot belong to their respective interval of probabilities on the IMC (*e.g.*, $p(t_2, t_1) = 0.8 \notin [0, 0.6] = P(s_2, s_1)$).

**Proposition 3.** *The* at-every-step *satisfaction relation is (strictly) more general than the* interval-markov-decision-process *satisfaction relation which is (strictly) more general than the* once-and-for-all *satisfaction relation.*

*Proof.* Let $\mathcal{I} = (S, s_0, P, V)$ be an IMC and $\mathcal{M} = (T, t_0, p, V')$ be an MC. We show that (1) $\mathcal{M} \models^o_{\mathtt{I}} \mathcal{I} \Rightarrow \mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$; (2) $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I} \Rightarrow \mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$; (3) in general $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I} \not\Rightarrow \mathcal{M} \models^o_{\mathtt{I}} \mathcal{I}$; (4) in general $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I} \not\Rightarrow \mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$. This will prove that $\models^a_{\mathtt{I}}$ is strictly more general than $\models^d_{\mathtt{I}}$ and that $\models^d_{\mathtt{I}}$ is strictly more general than $\models^o_{\mathtt{I}}$. At the same time, note that the following examples also illustrates that even if a Markov chain satisfies an IMC with the same graph representation w.r.t. the $\models^a_{\mathtt{I}}$ relation it may not verify the $\models^o_{\mathtt{I}}$ relation.

(1) If $\mathcal{M} \models^o_{\mathtt{I}} \mathcal{I}$ then by definition of $\models^o_{\mathtt{I}}$ we have that $T = S$, $t_0 = s_0$, $V(s) = V'(s)$ for all $s \in S$, and $p(s)(s') \in P(s, s')$ for all $s, s' \in S$. The mapping $\pi$ from $T = S$ to $S$ being the identity function is such as required by definition of $\models^d_{\mathtt{I}}$: $\pi(t_0) = t_0 = s_0$, $V'(s) = V(s) = V(\pi(s))$ for all state $s \in S$, and $p(s)(s') \in P(\pi(s), \pi(s'))$ since $P(\pi(s), \pi(s')) = P(s, s')$ and $p(s)(s') \in P(s, s')$ for all $s, s' \in S$. Thus, $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$.

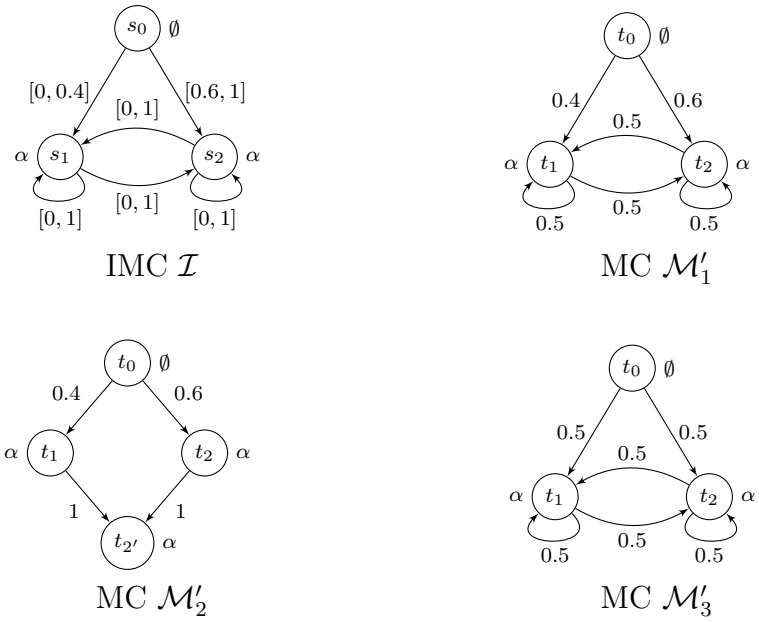



Figure 5.6: IMC $\mathcal{I}$, MCs $\mathcal{M}'_1$, $\mathcal{M}'_2$, and $\mathcal{M}'_3$ s.t. $\mathcal{M}'_1 \models^a_{\mathtt{I}} \mathcal{I}$, $\mathcal{M}'_1 \models^d_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}'_1 \models^o_{\mathtt{I}} \mathcal{I}$; $\mathcal{M}'_2 \models^d_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}'_2 \not\models^o_{\mathtt{I}} \mathcal{I}$; $\mathcal{M}'_3 \models^a_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}'_3 \not\models^d_{\mathtt{I}} \mathcal{I}$; the graph representation of $\mathcal{I}$, $\mathcal{M}'_1$, and $\mathcal{M}'_3$ are isomorphic;

(2) If $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$ then there exists a mapping $\pi$ from $T$ to $S$ s.t. $\pi(t_0) = s_0$, $V'(t) = V(\pi(t))$ for all state $t \in T$, and $p(t)(t') \in P(\pi(t), \pi(t'))$ for all pair of states $t, t'$ in $T$. The relation $\mathcal{R} = \{(t, \pi(t)) \mid t \in T\}$ is such as required by definition of $\models^a_{\mathtt{I}}$ (consider for each state in $T$ the correspondence function $\delta : T \to (S \to [0, 1])$ s.t. $\delta(t)(s) = 1$ if $\pi(t) = s$ and $\delta(t)(s) = 0$ otherwise). Thus $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$.

(3) Consider IMC $\mathcal{I}$ and MC $\mathcal{M}'_2$ from Figure 5.6. By definition of $\models^d_{\mathtt{I}}$ we have that $\mathcal{M}'_2 \models^d_{\mathtt{I}} \mathcal{I}$. Indeed, consider the mapping $\pi$ s.t. $\pi(t_0) = s_0$, $\pi(t_1) = s_1$, $\pi(t_2) = s_2$, and $\pi(t_{2'}) = s_2$. Let $p$ be the transition function of $\mathcal{M}'_2$ and $P$ be the interval probability transition function of $\mathcal{I}$. Clearly, we have that $p(t)(t') \in P(\pi(t), \pi(t'))$. On the other hand, it is clear that $\mathcal{M}'_2 \not\models^o_{\mathtt{I}} \mathcal{I}$ since $\mathcal{M}'_2$ and $\mathcal{I}$ do not share the same state space.

(4) Consider IMC $\mathcal{I}$ and MC $\mathcal{M}'_3$ from Figure 5.6. By definition of $\models^a_{\mathtt{I}}$ we have that $\mathcal{M}'_3 \models^a_{\mathtt{I}} \mathcal{I}$. Indeed, the relation $\mathcal{R}$ containing $(t_0, s_0)$, $(t_1, s_1)$, $(t_1, s_2)$, $(t_2, s_1)$ and $(t_2, s_2)$ is a satisfaction relation between $\mathcal{I}$ and $\mathcal{M}'_3$. Consider the correspondence function $\delta$ from $T$ to $(S \to [0, 1])$ such that $\delta(t_1)(s_1) = 4/5$, $\delta(t_1)(s_2) = 1/5$, $\delta(t_2)(s_2) = 1$, $\delta(t_0)(s_0) = 1$, and $\delta(t)(s) = 0$ otherwise. On the other hand, since the outgoing probabilities from state $t_0$ in $\mathcal{M}'_3$ do not belong to their respective interval on probabilities in $\mathcal{I}$, we have that $\mathcal{M}'_3 \not\models^d_{\mathtt{I}} \mathcal{I}$.

□

**Parametric Interval Markov Chains**, as introduced in [103], abstract IMCs by allowing (combinations of) parameters to be used as interval endpoints in IMCs. Under a

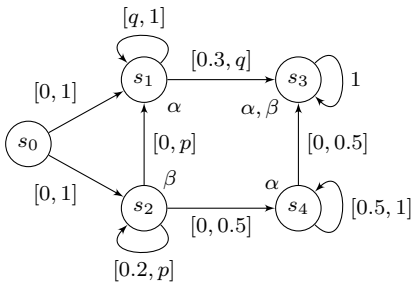



Figure 5.7: pIMC $\mathcal{P}$

given parameter valuation the pIMC yields an IMC as introduced above. pIMCs therefore allow the representation, in a compact way and with a finite structure, of a potentially infinite number of IMCs. Note that one parameter can appear in several transitions at once, requiring the associated transition probabilities to depend on one another. Let $Y$ be a finite set of parameters and $v$ be a valuation over $Y$. By combining notations used for IMCs and pMCs the set $\mathbb{I}(\mathbb{Q}_Y)$ contains all parametrized intervals over $[0,1]$, and for all $I = [f_1, f_2] \in \mathbb{I}(\mathbb{Q}_Y)$, $v(I)$ denotes the interval $[v(f_1), v(f_2)]$ if $0 \leq v(f_1) \leq v(f_2) \leq 1$ and the empty set otherwise[2]. We write $\texttt{pIMC}$ for the set containing all the parametric interval Markov chains.

**Definition 5.3.4** (Parametric Interval Markov Chain [103])**.** *A Parametric Interval Markov Chain (*pIMC *for short) is a tuple $\mathcal{P} = (S, s_0, P, V, Y)$, where $S$, $s_0$, $V$ and $Y$ are defined as for* pMC*s, and $P : S \times S \to \mathbb{I}(\mathbb{Q}_Y)$ associates with each potential transition a (parametric) interval.*

In [103] the authors introduced pIMCs where parametric interval endpoints are limited to linear combination of parameters. In our contribution we extend the pIMC model by allowing rational functions over parameters as endpoints of parametric intervals. Given a pIMC $\mathcal{P} = (S, s_0, P, V, Y)$ and a valuation $v$, we write $v(\mathcal{P})$ for the IMC $(S, s_0, P_v, V)$ obtained by replacing the transition function $P$ from $\mathcal{P}$ with the function $P_v : S \times S \to \mathbb{I}$ defined by $P_v(s, s') = v(P(s, s'))$ for all $s, s' \in S$. The IMC $v(\mathcal{P})$ is called an *instance* of pIMC $\mathcal{P}$. Finally, depending on the semantics chosen for IMCs, three satisfaction relations can be defined between MCs and pIMCs. They are written $\models^a_{\texttt{pI}}$, $\models^d_{\texttt{pI}}$, and $\models^o_{\texttt{pI}}$ and defined as follows: $\mathcal{M} \models^a_{\texttt{pI}} \mathcal{P}$ (resp. $\models^d_{\texttt{pI}}$, $\models^o_{\texttt{pI}}$) iff there exists an IMC $\mathcal{I}$ instance of $\mathcal{P}$ such that $\mathcal{M} \models^a_{\texttt{I}} \mathcal{I}$ (resp. $\models^d_{\texttt{I}}$, $\models^o_{\texttt{I}}$).

**Example 30.** Consider the pIMC $\mathcal{P} = (S, s_0, P, V, Y)$ given in Figure 5.7. The set of states $S$ and the labelling function are the same as in the MC and the IMC presented in Figures 5.1 and 5.3 respectively. The set of parameters $Y$ has two elements $p$ and $q$.

[2]Indeed, when $0 \leq v(f_1) \leq v(f_2) \leq 1$ is not respected, the interval is inconsistent and therefore empty.

Finally, the parametric intervals from the transition function $P$ are given by the edge labelling (*e.g.*, $P(s_1, s_3) = [0.3, q]$, $P(s_2, s_4) = [0, 0.5]$, and $P(s_3, s_3) = [1, 1]$). Note that the IMC $\mathcal{I}$ from Figure 5.3 is an instance of $\mathcal{P}$ (by assigning the value 0.6 to the parameter $p$ and 0.5 to $q$). Furthermore, as said in Example 29, the Markov Chains $\mathcal{M}_1$ and $\mathcal{M}_2$ (from Figures 5.1 and 5.5 respectively) satisfy $\mathcal{I}$ w.r.t. $\models^a_{\mathtt{I}}$, therefore $\mathcal{M}_1$ and $\mathcal{M}_2$ satisfy $\mathcal{P}$ w.r.t. $\models^a_{\mathtt{pI}}$.

In the following, we consider that the size of a pMC, IMC, or pIMC corresponds to its number of states plus its number of transitions not reduced to 0, $[0, 0]$ or $\emptyset$. We will also often need to consider the predecessors (`Pred`), and the successors (`Succ`) of some given states. Given a pIMC with a set of states $S$, a state $s$ in $S$, and a subset $S'$ of $S$, we write:

- $\mathtt{Pred}(s) = \{s' \in S \mid P(s', s) \notin \{\emptyset, [0, 0]\}\}$
- $\mathtt{Succ}(s) = \{s' \in S \mid P(s, s') \notin \{\emptyset, [0, 0]\}\}$
- $\mathtt{Pred}(S') = \bigcup_{s' \in S'} \mathtt{Pred}(s')$
- $\mathtt{Succ}(S') = \bigcup_{s' \in S'} \mathtt{Succ}(s')$

### 5.3.2 Abstraction Model Comparisons

IMC, pMC, and pIMC are three Markov chain Abstraction Models. In order to compare their expressiveness and compactness, we introduce the comparison operators $\sqsubseteq$ and $\equiv$. Let $(\mathtt{L}_1, \models_1)$ and $(\mathtt{L}_2, \models_2)$ be two Markov chain abstraction models containing respectively $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that $\mathcal{L}_1$ is entailed by $\mathcal{L}_2$, written $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$, iff all the MCs satisfying $\mathcal{L}_1$ satisfy $\mathcal{L}_2$ modulo bisimilarity. (*i.e.*, $\forall \mathcal{M} \models_1 \mathcal{L}_1, \exists \mathcal{M}' \models_2 \mathcal{L}_2$ s.t. $\mathcal{M}$ is bisimilar to $\mathcal{M}'$). Definition 5.3.5 recalls the bisimilarity property from [105]. We say that $\mathcal{L}_1$ is (semantically) equivalent to $\mathcal{L}_2$, written $\mathcal{L}_1 \equiv \mathcal{L}_2$, iff $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ and $\mathcal{L}_2 \sqsubseteq \mathcal{L}_1$. Definition 5.3.6 introduces succinctness based on the sizes of the abstractions.

**Definition 5.3.5** (Bisimulation [105])**.** *Let $\mathcal{M} = (S, S_0, p, V)$ be an* MC *possibly containing more than one initial state (*i.e.*, $S_0 \subseteq S$). A* probabilistic bisimulation *on $\mathcal{M}$ is an equivalence relation $\mathcal{R}$ on $S$ such that for all states $(s_1, s_2) \in \mathcal{R}$: $V(s_1) = V(s_2)$ and $\sigma_{t \in T} p(s_1, t) = \sigma_{t \in T} p(s_2, t)$ for all $T \in S/\mathcal{R}$. We say that two* MCs *$\mathcal{M}_1$ and $\mathcal{M}_2$ are bisimilar iff there exists a probabilistic bisimulation over their union containing the pair $(s_0, s'_0)$ where $s_0$ and $s'_0$) are respectively the initial state of $\mathcal{M}_1$ and $\mathcal{M}_2$.*

**Definition 5.3.6** (Succinctness)**.** *Let $(\mathtt{L}_1, \models_1)$ and $(\mathtt{L}_2, \models_2)$ be two Markov chain abstraction models. $\mathtt{L}_1$ is at least as succinct as $\mathtt{L}_2$, written $\mathtt{L}_1 \leq \mathtt{L}_2$, iff there exists a polynomial $p$ such that for every $\mathcal{L}_2 \in \mathtt{L}_2$, there exists $\mathcal{L}_1 \in \mathtt{L}_1$ s.t. $\mathcal{L}_1 \equiv \mathcal{L}_2$ and $|\mathcal{L}_1| \leq p(|\mathcal{L}_2|)$.*[3] *Thus, $\mathtt{L}_1$ is strictly more succinct than $\mathtt{L}_2$, written $\mathtt{L}_1 < \mathtt{L}_2$, iff $\mathtt{L}_1 \leq \mathtt{L}_2$ and $\mathtt{L}_2 \not\leq \mathtt{L}_1$.*

We start with a comparison of the succinctness of the pMC and IMC abstractions. Since pMCs allow the expression of dependencies between the probabilities assigned to distinct transitions while IMCs allow all transitions to be independant, it is clear that

[3] $|\mathcal{L}_1|$ and $|\mathcal{L}_2|$ are the sizes of $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively.

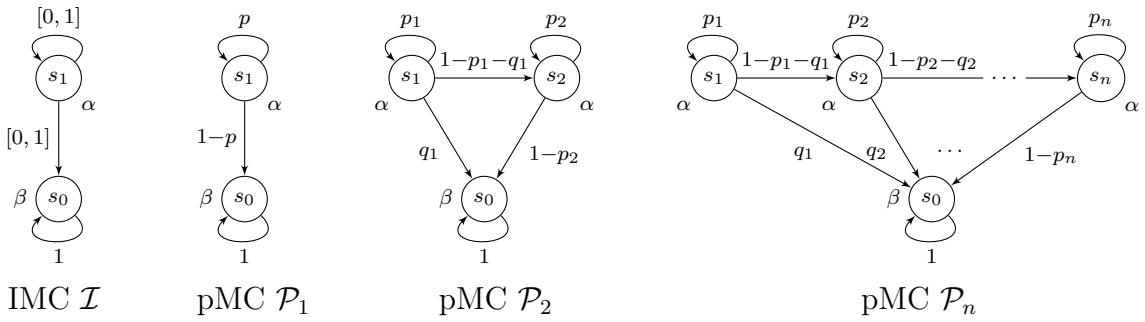



Figure 5.8: IMC $\mathcal{I}$ with three pMCs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_n$ entailed by $\mathcal{I}$ w.r.t. $\models^{a}_{\mathtt{I}}$.

there are pMCs without any equivalent IMCs (regardless of the IMC semantics used), therefore $(\mathtt{IMC}, \models^{\mathtt{o}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$, $(\mathtt{IMC}, \models^{\mathtt{d}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$, and $(\mathtt{IMC}, \models^{\mathtt{a}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$. On the other hand, IMCs imply that transition probabilities need to satisfy linear inequalities in order to fit given intervals. However, these types of constraints are not allowed in pMCs. It is therefore easy to exhibit IMCs that, regardless of the semantics considered, do not have any equivalent pMC specification. As a consequence, $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{o}}_{\mathtt{I}})$, $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{d}}_{\mathtt{I}})$, and $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{a}}_{\mathtt{I}})$.

We now compare pMCs and IMCs to pIMCs. Recall that the pIMC model is a Markov chain abstraction model allowing to declare parametric interval transitions, while the pMC model allows only parametric transitions (without intervals), and the IMC model allows interval transitions without parameters. Clearly, any pMC and any IMC can be translated into a pIMC with the right semantics (once-and-for-all for pMCs and the chosen IMC semantics for IMCs). This means that $(\mathtt{pIMC}, \models^{o}_{\mathtt{pI}})$ is more succinct than pMC and pIMC is more succinct than IMC for the three semantics. Furthermore, since pMC and IMC are not comparable due to the above results, we have that the pIMC abstraction model is strictly more succinct than the pMC abstraction model and than the IMC abstraction model with the right semantics. Our comparison results are presented in Proposition 4. Firstly, Lemma 1 states that IMC and pMC are not comparable w.r.t. satisfaction relations $\models^{o}_{\mathtt{I}}$, $\models^{d}_{\mathtt{I}}$, and $\models^{a}_{\mathtt{I}}$.

**Lemma 1.** pMC *and* IMC *abstraction models are not comparable in terms of succinctness: (1)* $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{a}}_{\mathtt{I}})$*, (2)* $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{d}}_{\mathtt{I}})$*, (3)* $\mathtt{pMC} \not\leq (\mathtt{IMC}, \models^{\mathtt{o}}_{\mathtt{I}})$*, (4)* $(\mathtt{IMC}, \models^{\mathtt{a}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$*, (5)* $(\mathtt{IMC}, \models^{\mathtt{d}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$*, and (6)* $(\mathtt{IMC}, \models^{\mathtt{o}}_{\mathtt{I}}) \not\leq \mathtt{pMC}$*.*

*Proof.* We give a sketch of proof for each statement. Let $(\mathtt{L}_1, \models_1)$ and $(\mathtt{L}_2, \models_2)$ be two Markov chain abstraction models. Recall that according to the succinctness definition (cf. Definition 5.3.6) $\mathtt{L}_1 \not\leq \mathtt{L}_2$ if there exists $\mathcal{L}_2 \in \mathtt{L}_2$ s.t. $\mathcal{L}_1 \not\equiv \mathcal{L}_2$ for all $\mathcal{L}_1 \in \mathtt{L}_1$.

(1) Consider IMC $\mathcal{I}$ and pMCs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_n$ (with $n \in \mathbb{N}$) from Figure 5.8. IMC $\mathcal{I}$ verifies the case (1). Indeed, the pMCs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_n$ (with $n \in \mathbb{N}$) are all entailed by $\mathcal{I}$ w.r.t. $\models^{a}_{\mathtt{I}}$ but none of them is equivalent to $\mathcal{I}$. Indeed one needs an infinite

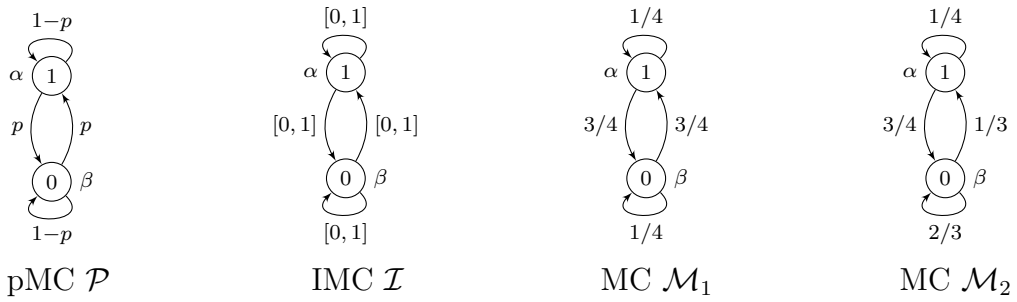



Figure 5.9: pMC $\mathcal{P}$, IMC $\mathcal{I}$, MC $\mathcal{M}_1$, and MC $\mathcal{M}_2$ s.t. $\mathcal{M}_1 \models_{\mathtt{p}} \mathcal{P}$ and $\mathcal{M}_1 \models^{a}_{\mathtt{I}} \mathcal{I}$ but $\mathcal{M}_2 \not\models_{\mathtt{p}} \mathcal{P}$ and $\mathcal{M}_2 \models^{a}_{\mathtt{I}} \mathcal{I}$ while $\mathcal{P}$ is entailed by $\mathcal{I}$ w.r.t. $\models^{a}_{\mathtt{I}}$.

countable number of states for creating a pMC equivalent to $\mathcal{I}$ w.r.t. $\models^{a}_{\mathtt{I}}$. However state spaces must be finite.

(2) Same example than from case (1) using Figure 5.8 can be used since all the pMCs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_n$ (with $n \in \mathbb{N}$) are entailed by the IMC $\mathcal{I}$ w.r.t. $\models^{d}_{\mathtt{I}}$

(3) Consider IMC $\mathcal{I}'$ similar to $\mathcal{I}$ from Figure 5.8 excepted that the transition from $s_1$ to $s_0$ is replaced by the interval $[0.5, 1]$. Since the pMC definition does not allow to bound values for parameters there is no equivalent $\mathcal{I}'$ w.r.t. $\models^{a}_{\mathtt{I}}$.

(4) Note that parameters in pMCs enforce transitions in the concrete MCs to receive the same value. Since parameters may range over continuous intervals there is no hope of modelling such set of Markov chains using a single IMC. Figure 5.9 illustrates this statement.

(5) Same remark than item (4)

(6) Same remark than item (4)

□

**Proposition 4.** *The Markov chain abstraction models can be ordered as follows w.r.t. succinctness:* $(\mathtt{pIMC}, \models^{o}_{\mathtt{pI}}) < (\mathtt{pMC}, \models_{\mathtt{p}})$, $(\mathtt{pIMC}, \models^{o}_{\mathtt{pI}}) < (\mathtt{IMC}, \models^{\mathtt{o}}_{\mathtt{I}})$, $(\mathtt{pIMC}, \models^{d}_{\mathtt{pI}}) < (\mathtt{IMC}, \models^{\mathtt{d}}_{\mathtt{I}})$, *and* $(\mathtt{pIMC}, \models^{a}_{\mathtt{pI}}) < (\mathtt{IMC}, \models^{\mathtt{a}}_{\mathtt{I}})$.

*Proof.* Recall that the pIMC model is a Markov chain abstraction model allowing to declare parametric interval transitions, while the pMC model allows only parametric transitions (without intervals), and the IMC model allows interval transitions without parameters. Clearly, any pMC and any IMC can be translated into a pIMC with the right semantics (once-and-for-all for pMCs and the chosen IMC semantics for IMCs). This means that $(\mathtt{pIMC}, \models^{o}_{\mathtt{pI}})$ is more succinct than pMC and that pIMC is more succinct than IMC for the three semantics. Furthermore since pMC and IMC are not comparable (cf Lemma 1), we have that the pIMC abstraction model is strictly more succinct than the pMC abstraction model and than the IMC abstraction model with the right semantics. □

Note that $(\texttt{pMC}, \models_{\texttt{p}}) \leq (\texttt{IMC}, \models^{\circ}_{\texttt{I}})$ could be achieved by considering a domain for each parameter of a pMC, which is not allowed here. However, this would not have any impact on our other results.

## 5.4 Qualitative Properties

As seen above, pIMCs are a succinct abstraction formalism for MCs. The aim of this section is to investigate qualitative properties for pIMCs, *i.e.*, properties that can be evaluated at the specification (pIMC) level, but that entail properties on its MC implementations. pIMC specifications are very expressive as they allow the abstraction of transition probabilities using both intervals and parameters. Unfortunately, as it is the case for IMCs, this allows the expression of incorrect specifications. In the IMC setting, this is the case either when some intervals are ill-formed or when there is no probability distribution matching the interval constraints of the outgoing transitions of some reachable state. In this case, no MC implementation exists that satisfies the IMC specification. Deciding whether an implementation that satisfies a given specification exists is called the *consistency* problem. In the pIMC setting, the consistency problem is made more complex because of the parameters which can also induce inconsistencies in some cases. One could also be interested in verifying whether there exists an implementation that reaches some target states/labels, and if so, propose a parameter valuation ensuring this property. This problem is called the consistent reachability problem. Both the consistency and the consistent reachability problems have already been investigated in the IMC and pIMC setting [109, 103]. In this section, we briefly recall these problems and propose new solutions based on CSP encodings. Our encodings are linear in the size of the original pIMCs whereas the algorithms from [109, 103] are exponential.

### 5.4.1 Existential Consistency

A pIMC $\mathcal{P}$ is existential consistent iff there exists an MC $\mathcal{M}$ satisfying $\mathcal{P}$ (*i.e.*, there exists an MC $\mathcal{M}$ satisfying an IMC $\mathcal{I}$ instance of $\mathcal{P}$). As seen in Section 5.2, pIMCs are equipped with three semantics: once-and-for-all ($\models^{o}_{\texttt{pI}}$), IMDP ($\models^{d}_{\texttt{pI}}$) and at-every-step ($\models^{a}_{\texttt{pI}}$). Recall that $\models^{o}_{\texttt{pI}}$ imposes that the underlying graph structure of implementations needs to be isomorphic to the graph structure of the corresponding specification. In contrast, $\models^{d}_{\texttt{pI}}$ and $\models^{a}_{\texttt{pI}}$ allows implementations to have a different graph structure. It therefore seems that some pIMCs could be inconsistent w.r.t $\models^{o}_{\texttt{pI}}$ while being consistent w.r.t $\models^{a}_{\texttt{pI}}$. On the other hand, checking the consistency w.r.t $\models^{o}_{\texttt{pI}}$ seems easier because of the fixed graph structure.

In [109], the author firstly proved that $\models^{a}_{\texttt{pI}}$ and $\models^{o}_{\texttt{pI}}$ semantics are equivalent w.r.t. existential consistency, and proposed a CSP encoding for verifying this property which is exponential in the size of the pIMC. Now, by Proposition 3 we also get that the three

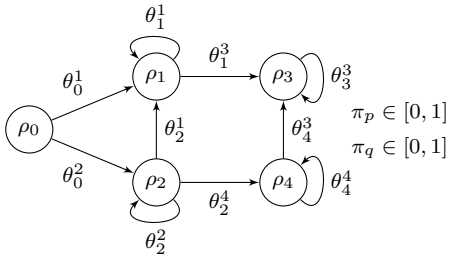



Figure 5.10: Variables in the CSP produced by $\mathbf{C}_{\exists\mathbf{c}}$ for the pIMC $\mathcal{P}$ from Fig. 5.7

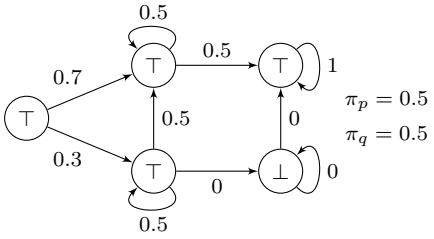



Figure 5.11: A solution to the CSP $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$ for the pIMC $\mathcal{P}$ from Fig. 5.7

semantics $\models^d_{\mathtt{pI}}$, $\models^a_{\mathtt{pI}}$, and $\models^o_{\mathtt{pI}}$ are equivalent w.r.t. existential consistency. Based on this result of semantics equivalence we propose a new CSP encoding, written $\mathbf{C}_{\exists\mathbf{c}}$, for verifying the existential consistency property for pIMCs.

Let $\mathcal{P} = (S,s_0,P,V,Y)$ be a pIMC, we write $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$ for the CSP produced by $\mathbf{C}_{\exists\mathbf{c}}$ according to $\mathcal{P}$. Any solution of $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$ will correspond to an MC satisfying $\mathcal{P}$. In $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$, we use one variable $\pi_p$ with domain $[0,1]$ per parameter $p$ in $Y$; one variable $\theta_s^{s'}$ with domain $[0,1]$ per transition $(s,s')$ in $\{\{s\} \times \mathtt{Succ}(s) \mid s \in S\}$; and one Boolean variable $\rho_s$ per state $s$ in $S$. These Boolean variables will indicate for each state whether it appears in the MC solution of the CSP (*i.e.*, in the MC satisfying the pIMC $\mathcal{P}$). For each state $s \in S$, Constraints are as follows:

**(1)** $\rho_s$, if $s = s_0$

**(2)** $\neg\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Pred}(s)\setminus\{s\}} \theta_{s'}^{s} = 0$, if $s \neq s_0$

**(3)** $\neg\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Succ}(s)} \theta_s^{s'} = 0$

**(4)** $\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Succ}(s)} \theta_s^{s'} = 1$

**(5)** $\rho_s \Rightarrow \theta_s^{s'} \in P(s,s')$, for all $s' \in \mathtt{Succ}(s)$

Recall that given a pIMC $\mathcal{P}$ the objective of the CSP $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$ is to construct an MC $\mathcal{M}$ satisfying $\mathcal{P}$. Constraint **(1)** states that the initial state $s_0$ appears in $\mathcal{M}$. Constraint **(2)** ensures that for each non-initial state $s$, variable $\rho_s$ is set to false iff $s$ is not reachable from its predecessors. Constraint **(4)** ensures that if a state $s$ appears in $\mathcal{M}$, then its outgoing transitions form a probability distribution. On the contrary, Constraint **(3)** propagates non-appearing states (*i.e.*, if a state $s$ does not appear in $\mathcal{M}$ then all its outgoing transitions are set to zero). Finally, Constraint **(5)** states that, for all appearing states, the outgoing transition probabilities must be selected inside the specified intervals.

**Example 31.** Consider the pIMC $\mathcal{P}$ given in Figure 5.7. Figure 5.10 describes the variables in $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$: one variable per transition (*e.g.*, $\theta_0^1$, $\theta_0^2$, $\theta_1^1$), one Boolean variable per state (*e.g.*, $\rho_0$, $\rho_1$), and one variable per parameter ($\pi_p$ and $\pi_q$). The following constraints correspond to the Constraints **(2)**, **(3)**, **(4)**, and **(5)** generated by our encoding $\mathbf{C}_{\exists\mathbf{c}}$ for the state 2 of $\mathcal{P}$:

$$\neg\rho_2 \Leftrightarrow \theta_0^2 = 0 \qquad \rho_2 \Leftrightarrow \theta_2^1 + \theta_2^2 + \theta_2^4 = 1 \qquad \rho_2 \Rightarrow 0.2 \leq \theta_2^2 \leq \pi_p$$
$$\neg\rho_2 \Leftrightarrow \theta_2^1 + \theta_2^2 + \theta_2^4 = 0 \qquad \rho_2 \Rightarrow 0 \leq \theta_2^1 \leq \pi_p \qquad \rho_2 \Rightarrow 0 \leq \theta_2^4 \leq 0.5$$

Finally, Figure 5.11 describes a solution for the CSP $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$. Note that given a solution of a pIMC encoded by $\mathbf{C}_{\exists\mathbf{c}}$, one can construct an MC satisfying the given pIMC w.r.t. $\models^o_{\mathtt{I}}$ by keeping all the states $s$ such that $\rho_s$ is equal to $\mathsf{true}$ and considering the transition function given by the probabilities in the $\theta_s^{s'}$ variables. We now show that our encoding works as expected.

**Proposition 5.** *A* pIMC $\mathcal{P}$ *is existential consistent iff* $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$ *is satisfiable.*

*Proof.* Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC.

[$\Rightarrow$] The CSP $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P}) = (X, D, C)$ is satisfiable implies that there exists a valuation $v$ of the variables in $X$ satisfying all the constraints in $C$. Consider the MC $\mathcal{M} = (S, s_0, p, V)$ such that $p(s, s') = v(\theta_s^{s'})$, for all $\theta_s^{s'} \in \Theta$ and $p(s, s') = 0$ otherwise.

Firstly, we show by induction that for any state $s$ in $S$: "if $s$ is reachable in $\mathcal{M}$ then $v(\rho_s)$ equals to $\mathsf{true}$". This is correct for the initial state $s_0$ thanks to the Constraint **(1)**. Let $s$ be a state in $S$ and assume that the property is correct for all its predecessors. By the Constraints **(2)**, $v(\rho_s)$ equals $\mathsf{true}$ if there exists at least one predecessor $s'' \neq s$ reaching $s$ with a non-zero probability (*i.e.*, $v(\theta_{s''}^s) \neq 0$). This is only possible by the Constraint **(4)** if $v(\rho_{s''})$ equals $\mathsf{true}$. Thus $v(\rho_s)$ equals $\mathsf{true}$ if there exists one reachable state $s''$ s.t. $v(\theta_{s''}^s) \neq 0$.

Secondly, we show that $\mathcal{M}$ satisfies the pIMC $\mathcal{P}$ w.r.t. $\models^o_{\mathtt{I}}$. We use Theorem 4 from [103] stating that $\models^a_{\mathtt{pI}}$ and $\models^o_{\mathtt{pI}}$ are equivalent w.r.t. qualitative reachability. We proved above that for all reachable states $s$ in $\mathcal{M}$, we have $v(\rho_s)$ equals to $\mathsf{true}$. By the Constraints **(5)** it implies that for all reachable states $s$ in $\mathcal{M}$: $p(s)(s') \in P(s, s')$ for all $s$ and $s'$.[4]

[$\Leftarrow$] The pIMC $\mathcal{P}$ is consistent implies by the Theorem 4 from [103] stating that $\models^a_{\mathtt{pI}}$ and $\models^o_{\mathtt{pI}}$ are equivalent w.r.t. qualitative reachability, that there exists an implementation of the form $\mathcal{M} = (S, s_0, p, V)$ where, for all reachable states $s$ in $\mathcal{M}$, it holds that $p(s)(s') \in P(s, s')$ for all $s'$ in $S$. Consider $\mathcal{M}' = (S, s_0, p', V)$ s.t. for each non reachable state $s$ in $S$: $p'(s)(s') = 0$, for all $s' \in S$. The valuation $v$ is s.t. $v(\rho_s)$ equals $\mathsf{true}$ iff $s$ is reachable in $\mathcal{M}$, $v(\theta_s^{s'}) = p'(s)(s')$, and for each parameter $y \in Y$ a valid value can be selected according to $p$ and $P$ when considering reachable states. Finally, by construction, $v$ satisfies the CSP $\mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$. □

Our existential consistency encoding is linear in the size of the pIMC instead of exponential for the encoding from [103] which enumerates the powerset of the states in the pIMC resulting in deep nesting of conjunctions and disjunctions.

[4] As illustrated in Example 31, $\mathcal{M}$ is not a well formed MC since some unreachable states do not respect the probability distribution property. However, one can correct it by simply setting one of its outgoing transition to 1 for each unreachable state.

### 5.4.2 Qualitative Reachability

Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC and $\alpha \subseteq A$ be a state label. We say that $\alpha$ is *existential reachable* in $\mathcal{P}$ iff there exists an implementation $\mathcal{M}$ of $\mathcal{P}$ where $\alpha$ is reachable (*i.e.*, $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) > 0$). In a dual way, we say that $\alpha$ is *universal reachable* in $\mathcal{P}$ iff $\alpha$ is reachable in any implementation $\mathcal{M}$ of $\mathcal{P}$. As for existential consistency, we use a result from [109] that states that the $\models^a_{\mathtt{I}}$ and the $\models^o_{\mathtt{I}}$ pIMC semantics are equivalent w.r.t. existential (and universal) reachability. As for the consistency problem, we get by Proposition 3 that the three IMC semantics are equivalent w.r.t. existential (and universal) reachability. Note first that in our $\mathbf{C}_{\exists\mathbf{r}}$ encoding each $\rho_s$ variable indicates if the state $s$ appears in the constructed Markov chain. However, the $\rho_s$ variable does not indicate if the state $s$ is reachable from the initial state, but only if it is reachable from at least one other state (*i.e.*, possibly different from $s_0$). Indeed, if the graph representation of the constructed Markov chain has strongly connected components (SCCs for short), then all the $\rho_s$ variables in one SCC may be set to $\mathsf{true}$ while this SCC may be unreachable from the initial state. This is not an issue in the case of the consistency problem. Indeed, if a Markov chain containing an unreachable SCC is proved consistent, then it is also consistent without this unreachable SCC. However, in the case of the reachability problem, these SCCs are an issue. The following encoding therefore takes into account these isolated SCCs such that $\rho_s$ variables are set to $\mathsf{true}$ if and only if they are all reachable from the initial state. This encoding will solve the qualitative reachability problems (*i.e.*, checking qualitative reachability from the initial state). We propose a new CSP encoding, written $\mathbf{C}_{\exists\mathbf{r}}$, that extends $\mathbf{C}_{\exists\mathbf{c}}$, for verifying these properties. Formally, CSP $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P}) = (X \cup X', D \cup D', C \cup C')$ is such that $(X, D, C) = \mathbf{C}_{\exists\mathbf{c}}(\mathcal{P})$, $X'$ contains one integer variable $\omega_s$ with domain $[0, |S|]$ per state $s$ in $S$, $D'$ contains the domains of these variables, and $C'$ is composed of the following constraints for each state $s \in S$:

**(6)** $\omega_s = 1$, if $s = s_0$

**(7)** $\omega_s \neq 1$, if $s \neq s_0$

**(8)** $\rho_s \Leftrightarrow (\omega_s \neq 0)$

**(9)** $\omega_s > 1 \Rightarrow \bigvee_{s' \in \mathtt{Pred}(s) \setminus \{s\}} (\omega_s = \omega_{s'} + 1) \wedge (\theta_s^{s'} > 0)$, if $s \neq s_0$

**(10)** $\omega_s = 0 \Leftrightarrow \bigwedge_{s' \in \mathtt{Pred}(s) \setminus \{s\}} (\omega_{s'} = 0) \vee (\theta_s^{s'} = 0)$, if $s \neq s_0$

Recall first that CSP $\mathbf{C}_{\exists\mathbf{c}}(P)$ constructs a Markov chain $\mathcal{M}$ satisfying $\mathcal{P}$ w.r.t. $\models^o_{\mathtt{I}}$. Informally, for each state $s$ in $\mathcal{M}$ the Constraints **(6)**, **(7)**, **(9)** and **(10)** in $\mathbf{C}_{\exists\mathbf{r}}$ ensure that $\omega_s = k$ iff there exists in $\mathcal{M}$ a path from the initial state to $s$ of length $k - 1$ with non zero probability; and state $s$ is not reachable in $\mathcal{M}$ from the initial state $s_0$ iff $\omega_s$ equals to 0. Finally, Constraint **(8)** enforces the Boolean reachability indicator variable $\rho_s$ to bet set to $\mathsf{true}$ iff there exists a path with non zero probability in $\mathcal{M}$ from the initial state $s_0$ to $s$ (*i.e.*, $\omega_s \neq 0$).

Let $S_\alpha$ be the set of states from $\mathcal{P}$ labeled with $\alpha$. $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$ therefore produces a Markov chain satisfying $\mathcal{P}$ where reachable states $s$ are such that $\rho_s = \mathsf{true}$. As a consequence, $\alpha$ is existential reachable in $\mathcal{P}$ iff $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$ admits a solution such that $\bigvee_{s\in S_\alpha} \rho_s$; and $\alpha$ is universal reachable in $\mathcal{P}$ iff $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$ admits no solution such that $\bigwedge_{s\in S_\alpha} \neg\rho_s$. This is formalised in the following proposition.

**Proposition 6.** *Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a* pIMC*, $\alpha \subseteq A$ be a state label, $S_\alpha = \{s \mid V(s) = \alpha\}$, and $(X, D, C)$ be the* CSP *$\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$.*

- CSP $(X, D, C \cup \bigvee_{s\in S_\alpha} \rho_s)$ *is satisfiable iff $\alpha$ is existential reachable in $\mathcal{P}$*
- CSP $(X, D, C \cup \bigwedge_{s\in S_\alpha} \neg\rho_s)$ *is unsatisfiable iff $\alpha$ is universal reachable in $\mathcal{P}$*

*Proof.* Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC, $\alpha \subseteq A$ be a state label, $S_\alpha = \{s \mid V(s) = \alpha\}$, and $(X, D, C)$ be the CSP $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$. Recall first, that by Proposition 5 the constraints **(1)** to **(5)** in $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$ are satisfied iff there exists an MC $\mathcal{M}$ satisfying $\mathcal{P}$ w.r.t. $\models^a_{\mathtt{I}}$.

- [$\Rightarrow$] If CSP $(X, D, C \cup \bigvee_{s\in S_\alpha} \rho_s)$ is satisfiable then there exists a valuation $v$ solution of this CSP and a corresponding MC $\mathcal{M}$ satisfying $\mathcal{P}$ w.r.t. $\models^a_{\mathtt{I}}$ such as presented in the proof of Proposition 5. Furthermore, the constraints **(6)** to **(10)** ensure by induction that for all state $s \in S$: $v(\omega_s) = k$ with $k \geq 1$ if there exists a path from the initial state $s_0$ to the state $s$ of size $k-1$ with non zero probability in $\mathcal{M}$, and $v(\omega_s) = 0$ otherwise. By constraint **(8)** we have that $v(\rho_s) = \mathsf{true}$ iff state $s$ is reachable in $\mathcal{M}$ from the initial state $s_0$. Finally, constraint $\bigvee_{s\in S_\alpha} \rho_s$ ensures that at least one state labeled with $\alpha$ must be reachable in $\mathcal{M}$. Thus, $\alpha$ is existential reachable in $\mathcal{P}$.

  [$\Leftarrow$] If $\alpha$ is existential reachable in $\mathcal{P}$, then by [103] there exists an MC $\mathcal{M}$ satisfying $\mathcal{P}$ w.r.t. $\models^o_{\mathtt{I}}$ s.t. $\alpha$ is reachable in $\mathcal{M}$. By construction of our encoding, one can easily construct from $\mathcal{M}$ a valuation $v$ satisfying all the constraints in $C \cup \bigvee_{s\in S_\alpha} \rho_s$ s.t. $v(\omega_s)$ contains the size (plus one) of an existing path in $\mathcal{M}$ from the initial state to the state $s$ with a non zero probability, and $v(\omega_s) = 0$ if $s$ is not reachable in $\mathcal{M}$.

- Note that $\alpha$ is universal reachable in $\mathcal{P}$ iff there is no MC $\mathcal{M}$ satisfying $\mathcal{P}$ w.r.t. $\models^a_{\mathtt{I}}$ s.t. none of the states labelled with $\alpha$ is reachable in $\mathcal{M}$. "CSP $(X, D, C \cup \bigwedge_{s\in S_\alpha} \neg\rho_s)$ is unsatisfiable" encodes this statement.

□

As for the existential consistency problem, we have an exponential gain in terms of size of the encoding compared to [103]: the number of constraints and variables in $\mathbf{C}_{\exists\mathbf{r}}$ is linear in terms of the size of the encoded pIMC.

**Remark.** In $\mathbf{C}_{\exists\mathbf{r}}$ Constraints **(2)** inherited from $\mathbf{C}_{\exists\mathbf{c}}$ are entailed by Constraints **(8)** and **(10)** added to $\mathbf{C}_{\exists\mathbf{r}}$. Thus, in a practical approach one may ignore Constraints **(2)** from $\mathbf{C}_{\exists\mathbf{c}}$ if they do not improve the solver performances.

## 5.5 Quantitative Properties

We now move to the verification of quantitative reachability properties in pIMCs. Quantitative reachability has already been investigated in the context of pMCs and IMCs with the once-and-for-all semantics. In this section, we propose our main theoretical contribution: a theorem showing that the three IMC semantics are equivalent with respect to quantitative reachability, which allows the extension of all results from [107, 102] to the at-every-step semantics. Based on this result, we also extend the CSP encodings introduced in Section 5.4 in order to solve quantitative reachability properties on pIMCs regardless of their semantics.

### 5.5.1 Equivalence of $\models^o_{\mathtt{I}}$, $\models^d_{\mathtt{I}}$ and $\models^a_{\mathtt{I}}$ w.r.t quantitative reachability

Given an IMC $\mathcal{I} = (S, s_0, P, V)$ and a state label $\alpha \subseteq A$, a quantitative reachability property on $\mathcal{I}$ is a property of the type $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$, where $0 < p < 1$ and $\sim \in \{\leq, <, >, \geq\}$. Such a property is verified iff there exists an MC $\mathcal{M}$ satisfying $\mathcal{I}$ (with the chosen semantics) such that $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha)\sim p$.

As explained above, existing techniques and tools for verifying quantitative reachability properties on IMCs only focus on the once-and-for-all and the IMDP semantics. However, to the best of our knowledge, there are no works addressing the same problem with the at-every-step semantics or showing that addressing the problem in the once-and-for-all and IMDP setting is sufficiently general. The following theorem fills this theoretical gap by proving that the three IMC semantics are equivalent w.r.t quantitative reachability. In other words, for all MC $\mathcal{M}$ such that $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$ or $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$ and for all state label $\alpha$, there exist MCs $\mathcal{M}_{\leq}$ and $\mathcal{M}_{\geq}$ such that $\mathcal{M}_{\leq} \models^o_{\mathtt{I}} \mathcal{I}$, $\mathcal{M}_{\geq} \models^o_{\mathtt{I}} \mathcal{I}$ and $\mathbb{P}^{\mathcal{M}_{\leq}}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}_{\geq}}(\Diamond\alpha)$. This is formalised in the following theorem.

**Theorem 1.** *Let $\mathcal{I} = (S, s_0, P, V)$ be an* IMC*, $\alpha \subseteq A$ be a state label, $\sim \in \{\leq, <, >, \geq\}$ and $0 < p < 1$. $\mathcal{I}$ satisfies $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ with the once-and-for-all semantics iff $\mathcal{I}$ satisfies $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ with the IMDP semantics iff $\mathcal{I}$ satisfies $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ with the at-every-step semantics.*

The proof presented in the following is constructive: we use the structure of the relation $\mathcal{R}$ from the definition of $\models^a_{\mathtt{I}}$ in order to build the MCs $\mathcal{M}_{\leq}$ and $\mathcal{M}_{\geq}$.

In the following, when it is not specified the IMC satisfaction relation considered is the *at-every-step* semantics (*i.e.*, the $\models^a_{\mathtt{I}}$ satisfaction relation). As said previously, we use the structure of the relation $\mathcal{R}$ from the definition of $\models^a_{\mathtt{I}}$ in order to build the MCs $\mathcal{M}_{\leq}$ and $\mathcal{M}_{\geq}$ presented in Theorem 1. Thus, we introduce some notations relative to $\mathcal{R}$. Let $\mathcal{I} = (S, s_0, P, V^I)$ be an IMC and $\mathcal{M} = (T, t_0, p, V^M)$ be an MC such that $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$. Let

$\mathcal{R} \subseteq T \times S$ be a satisfaction relation between $\mathcal{M}$ and $\mathcal{I}$. For all $t \in T$ we write $\mathcal{R}(t)$ for the set $\{s \in S \mid t\ \mathcal{R}\ s\}$, and for all $s \in S$ we write $\mathcal{R}^{-1}(t)$ for the set $\{t \in T \mid s\ \mathcal{R}\ t\}$. Furthermore we say that $\mathcal{M}$ satisfies $\mathcal{I}$ with degree $n$ (written $\mathcal{M} \models^{n_a}_{\mathtt{I}} \mathcal{I}$) if $\mathcal{M}$ satisfies $\mathcal{I}$ with a satisfaction relation $\mathcal{R}$ such that each state $t \in T$ is associated by $\mathcal{R}$ to at most $n$ states from $S$ (*i.e.*, $|\mathcal{R}(t)| \leq n$); $\mathcal{M}$ satisfies $\mathcal{I}$ with the same structure than $\mathcal{I}$ if $\mathcal{M}$ satisfies $\mathcal{I}$ with a satisfaction relation $\mathcal{R}$ such that each state $t \in T$ is associated to at most one state from $S$ and each state $s \in S$ is associated to at most one state from $T$ (*i.e.*, $|\mathcal{R}(t)| \leq 1$ for all $t \in T$ and $|\mathcal{R}^{-1}(s)| \leq 1$ for all $s \in S$).

**Proposition 7.** *Let $\mathcal{I}$ be an* IMC. *If an* MC $\mathcal{M}$ *satisfies $\mathcal{I}$ with degree $n \in \mathbb{N}$ then there exists an* MC $\mathcal{M}'$ *satisfying $\mathcal{I}$ with degree* 1 *such that $\mathcal{M}$ and $\mathcal{M}'$ are bisimilar.*

The main idea for proving Proposition 7 is that if an MC $\mathcal{M}$ with states space $T$ satisfies an IMC $\mathcal{I}$ with a states space $S$ according to a satisfaction relation $\mathcal{R}$ then, each state $t$ related by $\mathcal{R}$ to many states $s_1, \ldots, s_n$ (with $n > 1$) can be split in $n$ states $t^1, \ldots, t^n$. The derived MC will satisfy $\mathcal{I}$ with a satisfaction relation $\mathcal{R}'$ where each $t_i$ is only associated by $\mathcal{R}'$ to the state $s_i$ ($i \leq n$). This $\mathcal{M}'$ will be bisimilar to $\mathcal{M}$ and it will satisfy $\mathcal{I}$ with degree 1. Note that by construction the size of the resulting MC is in $O(|\mathcal{M}| \times |\mathcal{I}|)$.

Furthermore, we will use the *until* temporal modality (abbreviated U ) as presented in [105]. Let $\mathcal{M}$ be an MC and $\alpha$, $\beta$ be two state labelings. The probability of the property $\alpha$ U $\beta$ is given by the sum of the probabilities of all the finite paths starting in the initial state containing only states labeled with $\alpha$ excepted for the last state which is labeled with $\beta$. Formally, let $\mathsf{until}_{s_0}(s) = \{\omega \in S^* \mid \omega = s_0, \ldots s_n$ with $V(s_n) = \beta$ and $V(s_i) = \alpha\ \forall 0 \leq i < n\}$ be the set of such paths. Thus, $\mathbb{P}^{\mathcal{M}}(\alpha\ \mathrm{U}\ \beta) = \sum_{\omega \in \mathsf{reach}_{s_0}(s)} \mathbb{P}^{\mathcal{M}}(\omega)$. As for the reachability property, this notation naturally extends to states instead of labels, as well as conjunctions and disjunctions of states/labels.

*Proof for Proposition 7.* Let $\mathcal{I} = (S, s_0, P, V^I)$ be an IMC and $\mathcal{M} = (T, t_0, p, V)$ be an MC. If $\mathcal{M}$ satisfies $\mathcal{I}$ (with degree $n$) then there exists a satisfaction relation $\mathcal{R}$ verifying the $\models^a_{\mathtt{I}}$ satisfaction relation. For each association $(t, s) \in \mathcal{R}$, we write $\delta^s_t$ the correspondence function chosen for this pair of states. $\mathcal{M}$ satisfies $\mathcal{I}$ with degree $n$ means that each state in $\mathcal{M}$ is associated by $\mathcal{R}$ to at most $n$ states in $\mathcal{I}$. To construct an MC $\mathcal{M}'$ satisfying $\mathcal{I}$ with degree 1 we create one state in $\mathcal{M}'$ per association $(t, s)$ in $\mathcal{R}$. Formally, let $\mathcal{M}'$ be equal to $(U, u_0, p', V')$ such that $U = \{u^s_t \mid (t, s) \in \mathcal{R}\}$, $u_0 = u^{s_0}_{t_0}$, $V' = \{(u^s_t, v) \mid v = V(t)\}$, and $p'(u^s_t)(u^{s'}_{t'}) = p(t)(t') \times \delta^s_t(t')(s')$. The following computation shows that the outgoing probabilities given by $p'$ form a probability distribution for each state in $\mathcal{M}'$ and thus that $\mathcal{M}'$ is an MC.

$$\sum_{t'\mathcal{R}s'} p'(u_t^s)(u_{t'}^{s'}) = \sum_{t'\mathcal{R}s'} p(t)(t') \times \delta_t^s(t')(s')$$
$$= \sum_{t'\in T} p(t)(t') \times \sum_{s'\in S} \delta_t^s(t')(s') = \sum_{t'\in T} p(t)(t') \times 1 = 1$$

Finally, by construction of $\mathcal{M}'$ based on $\mathcal{M}$ which satisfies $\mathcal{I}$, we get that $\mathcal{R}' = \{(u_t^s, s) \mid t \in T, s \in S\}$ is a satisfaction relation between $\mathcal{M}'$ and $\mathcal{I}$. Furthermore $|\{s \mid (u, s) \in \mathcal{R}'\}|$ equals at most one. Thus, we get that $\mathcal{M}'$ satisfies $\mathcal{I}$ with degree 1.

Consider the relation $\mathcal{B}' = \{(u_t^s, t) \subseteq U \times T \mid (t, s) \in \mathcal{R}\}$. We note $\mathcal{B}$ the closure of $\mathcal{B}'$ by transitivity, reflexivity, and symmetry (*i.e.*, $\mathcal{B}$ is the minimal equivalence relation based on $\mathcal{B}'$). We prove that $\mathcal{B}$ is a bisimulation relation between $\mathcal{M}$ and $\mathcal{M}'$. By construction each equivalence class from $\mathcal{B}$ contains exactly one state $t$ from $T$ and all the states $u_t^s$ such that $(t, s) \in \mathcal{R}$. Let $(u_t^s, t)$ be in $\mathcal{B}$, $t'$ be a state in $T$, and $B$ be the equivalence class from $\mathcal{B}$ containing $t'$ (*i.e.*, $B$ is the set $\{t'\} \cup \{u_{t'}^{s'} \in U \mid s' \in S \text{ and } (t', s') \in \mathcal{R}\}$). Firstly note that by construction the labels agree on $u_t^s$ and $t$: $V'(u_t^s) = V(t)$. Secondly the following computation shows that $p'(u_t^s)(B \cap U)$ equals to $p(t)(B \cap T)$ and thus that $u_t^s$ and $t$ are bisimilar:

$$\begin{aligned} p'(u_t^s)(B \cap U) &= \sum_{u_{t'}^{s'} \in B \cap U} p'(u_t^s)(u_{t'}^{s'}) = \sum_{u_{t'}^{s'} \in B \cap U} p(t)(t') \times \delta_t^s(t')(s') \\ &= \sum_{\{s' \in S \mid s'\mathcal{R}t'\}} p(t)(t') \times \delta_t^s(t')(s') = p(t)(t') \times \sum_{\{s' \in S \mid s'\mathcal{R}t'\}} \delta_t^s(t')(s') \\ &= p(t)(t') \times 1 = p(t)(\{t'\}) = p(t)(B \cap T) \end{aligned}$$

□

**Corollary 1.** *Let $\mathcal{I}$ be an* IMC*, $\mathcal{M}$ be an* MC *satisfying $\mathcal{I}$, and $\gamma$ be a PCTL* formulae. There exists an* MC *$\mathcal{M}'$ satisfying $\mathcal{I}$ with degree* 1 *such that the probability $\mathbb{P}^{\mathcal{M}'}(\gamma)$ equals the probability $\mathbb{P}^{\mathcal{M}}(\gamma)$.*

Corollary 1 is derived from Proposition 7 joined with the probability preservation of the PCTL* formulae on bisimilar Markov chains (see [105], Theorem 10.67, p.813). Corollary 1 allows to reduce to one the number of states in the pIMC $\mathcal{I}$ satisfied by each state in the MC $\mathcal{M}$ while preserving probabilities. Thus, one can construct from an MC $\mathcal{M}$ satisfying an IMC $\mathcal{I}$ another MC $\mathcal{M}'$ satisfying the same IMC $\mathcal{I}$ where the states in $\mathcal{M}'$ are related to at most one state in $\mathcal{I}$. However, some states in $\mathcal{I}$ may still be related to many states in $\mathcal{M}'$. The objective of Lemma 2 is to reduce these relations to an "at most one" in both directions ($\mathcal{I}$ to $\mathcal{M}'$ and $\mathcal{M}'$ to $\mathcal{I}$).

**Lemma 2.** *Let $\mathcal{I} = (S, s_0, P, V)$ be an* IMC*, $\mathcal{M} = (T, t_0, p, V)$ be an* MC *satisfying $\mathcal{I}$ with degree* 1*, and $\alpha \subseteq A$ be a proposition. If $\mathcal{M}$ does not have the same structure than*

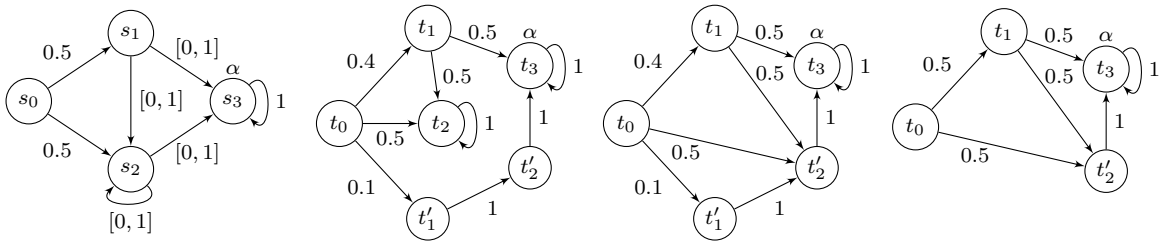



Figure 5.12: An IMC $\mathcal{I}$ and three MCs $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ satisfying $\mathcal{I}$ w.r.t. $\models^a_{\mathrm{I}}$ s.t. $\mathbb{P}^{\mathcal{M}_1}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}_2}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}_3}(\Diamond\alpha)$ and $\mathcal{M}_3$ has the same structure as $\mathcal{I}$.

*$\mathcal{I}$ then there exists an* MC $\mathcal{M}_1$ *(resp. $\mathcal{M}_2$) satisfying $\mathcal{I}$ with a set of states $S_1$ (resp. $S_2$) s.t. $S_1 \subset S$ and $\mathbb{P}^{\mathcal{M}_1}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$ (resp. $S_2 \subset S$ and $\mathbb{P}^{\mathcal{M}_2}(\Diamond\alpha) \geq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$).*

Lemma 2 reduces the number of states in $\mathcal{M}$ while preserving the maximal or minimal reachability probability. This lemma has a constructive proof. The main idea of the proof is that we select one state $s$ from the IMC $\mathcal{I}$ which is satisfied by many states $t_1, \ldots, t_n$ in $\mathcal{M}$. Thus, the MC $\mathcal{M}'$ keeping the state $t_k$ maximizing the probability of reaching $\alpha$ in $\mathcal{M}$ and removing all the other states $t_i$ (*i.e.*, remove the states $t_i$ such that $i \neq k$ and move the transitions to a state $t_i$ such that $i \neq k$ to arrive to the state $t_k$) will have less states than $\mathcal{M}$ and will verify $\mathbb{P}^{\mathcal{M}_1}(\Diamond\alpha) \geq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$. Figure 5.12 contains an IMC $\mathcal{I}$ and three MCs $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. This illustrates how Lemma 2 operates for reducing the state space. We describe how to obtain $\mathcal{M}_2$ from $\mathcal{M}_1$. Consider the state $s_2$ from $\mathcal{I}$. This state is related to the states $t_2$ and $t'_2$ in $\mathcal{M}_1$. Since $\mathbb{P}^{\mathcal{M}_1}_{t_2}(\Diamond\alpha) = 0$ and $\mathbb{P}^{\mathcal{M}_1}_{t'_2}(\Diamond\alpha) = 1$ we remove $t_2$ and we keep $t'_2$ which has an higher probability to reach $\alpha$. Then, all the transitions going to $t_2$ are changed in order to go to $t'_2$. This creates $\mathcal{M}_2$. Next, the same mechanism can be iterated to produce $\mathcal{M}_3$: consider $s_1$ from $\mathcal{I}$ and remove $t'_1$ and keep $t_1$ from $\mathcal{M}_2$ to produce $\mathcal{M}_3$. This allows to reduce the number of states in the constructed Markov chain while preserving the maximal/minimal reachability probability. Before proving Lemma 2, we introduce Lemma 3 which will be used for proving Lemma 2.

**Lemma 3.** *Let $\mathcal{M} = (S, s_0, p, V)$ be an* MC*, $\alpha \subseteq A$ be a proposition, and $s$ be a state from $S$. Then*

$$\mathbb{P}^{\mathcal{M}}_s(\Diamond\alpha) = \frac{\mathbb{P}^{\mathcal{M}}_s(\neg s \ \mathrm{U} \ \alpha)}{1 - \mathbb{P}^{\mathcal{M}}_s(\neg\alpha \ \mathrm{U} \ s)}$$

*Proof.* Let $S'$ be the subset of $S$ containing all the states labeled with $\alpha$ in $\mathcal{M}$. We write $\Omega_n$ with $n \in \mathbb{N}^*$ the set containing all the paths $\omega$ starting from $s$ s.t. state $s$ appears exactly $n$ times in $\omega$ and no state in $\omega$ is labeled with $\alpha$. Formally $\Omega_n$ contains all the $\omega = s_1, \ldots, s_k \in S^k$ s.t. $k \in \mathbb{N}$, $s_1$ is equal to $s$, $|\{i \in [1,k] \mid s_i = s\}| = n$, and $\alpha \not\subseteq V(s_i)$ for all $i \in [1,k]$. Given two sets of paths $\Omega$ and $\Omega'$, we write $\Omega \times \Omega'$ their Cartesian product which is the set of paths $\{\omega\omega' \mid \omega \in \Omega \text{ and } \omega' \in \Omega'\}$. We get by (a) that $(\mathbb{P}^{\mathcal{M}}_s(\Omega_n \times S'))_{n\geq 1}$ is a geometric series. For $(*)$ recall that given an MC $\mathcal{M}$ and two non-empty paths $\omega$ and

$\omega'$ on $\mathcal{M}$ s.t. $s$ and $s'$ are respectively the first state in $\omega$ and $\omega'$ we have by definition that $\mathbb{P}^{\mathcal{M}}_s(\omega\omega') = \mathbb{P}^{\mathcal{M}}_s(\omega s') \cdot \mathbb{P}^{\mathcal{M}}_{s'}(\omega')$. In (b) we partition the paths reaching $\alpha$ according to the $\Omega_n$ sets and we use the geometric series of the probabilities to retrieve the required result.

$$\begin{aligned}
\text{(a) } \mathbb{P}^{\mathcal{M}}_s(\Omega_n \times S') &= \mathbb{P}^{\mathcal{M}}_s(\Omega_1 \times \Omega_{n-1} \times S') \\
&\overset{(*)}{=} \mathbb{P}^{\mathcal{M}}_s(\Omega_1 \times \{s\}) \cdot \mathbb{P}^{\mathcal{M}}_s(\Omega_{n-1} \times S') \\
&= \mathbb{P}^{\mathcal{M}}_s(\neg\alpha \text{ U } s) \cdot \mathbb{P}^{\mathcal{M}}_s(\Omega_{n-1} \times S')
\end{aligned}$$

$$\begin{aligned}
\text{(b) } \mathbb{P}^{\mathcal{M}}_s(\Diamond\alpha) &= \Sigma^{+\infty}_{n=1}\mathbb{P}^{\mathcal{M}}_s(\Omega_n \times S') \\
&= \frac{\mathbb{P}^{\mathcal{M}}_s(\Omega_1 \times S')}{1 - \mathbb{P}^{\mathcal{M}}_s(\neg\alpha \text{ U } s)} \\
&= \frac{\mathbb{P}^{\mathcal{M}}_s(\neg s \text{ U } \alpha)}{1 - \mathbb{P}^{\mathcal{M}}_s(\neg\alpha \text{ U } s)}
\end{aligned}$$

□

*Proof for Lemma 2.* Let $\mathcal{I} = (S, s_0, P, V^I)$ be an IMC and $\mathcal{M} = (T, t_0, p, V)$ be an MC satisfying $\mathcal{I}$ with degree 1. We write $\mathcal{R}$ the satisfaction relation between $\mathcal{M}$ and $\mathcal{I}$ with degree 1. The following proves in 3 steps the $\mathbb{P}^{\mathcal{M}_1}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$ case.

1. We would like to construct an MC $\mathcal{M}'$ satisfying $\mathcal{I}$ with less states than $\mathcal{M}'$ such that $\mathbb{P}^{\mathcal{M}'}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$. Since the degree of $\mathcal{R}$ equals to 1 each state $t$ in $T$ is associated to at most one state $s$ in $S$. Furthermore, since $\mathcal{M}$ does not have the same structure than $\mathcal{I}$ then there exists at most one state from $S$ which is associated by $\mathcal{R}$ to many states from $T$. Let $\bar{s}$ be a state from $S$ such that $|\mathcal{R}^{-1}(s)| \geq 2$, $\bar{T} = \{t_1, \ldots, t_n\}$ be the set $\mathcal{R}^{-1}(s)$ where the $t_i$ are ordered by decreasing probability of reaching $\alpha$ (*i.e.*, $\mathbb{P}^{\mathcal{M}}_{t_i}(\Diamond\alpha) \geq \mathbb{P}^{\mathcal{M}}_{t_{i+1}}(\Diamond\alpha)$ for all $1 \leq i < n$). In the following we refer $\bar{t}$ as $t_n$. We produce $\mathcal{M}'$ from $\mathcal{M}$ by replacing all the transitions going to a state $t_1, \ldots, t_{n-1}$ by a transition going to $t_n$, and by removing the corresponding states. Formally $\mathcal{M}' = (T', t_0, p', V')$ s.t. $T' = (T \setminus \bar{T}) \cup \{\bar{t}\}$, $V'$ is the restriction of $V$ on $T'$, and for all $t, t' \in T'$: $p'(t)(t') = p(t)(t')$ if $t' \neq \bar{t}$ and $p'(t)(t') = \sum_{t' \in \bar{T}} p(t)(t')$ otherwise.

$$\begin{aligned}\sum_{t'\in T'} p'(t)(t') &= \sum_{t'\in T'\setminus\{\bar{t}\}} p'(t)(t') \;+\; p'(t)(\bar{t}) \\ &\overset{(1)}{=} \sum_{t'\in T'\setminus\{\bar{t}\}} p(t)(t') \;+\; \sum_{t'\in \bar{T}} p(t)(t') \\ &= \sum_{t'\in T'\setminus\{\bar{t}\}\cup\bar{T}} p(t)(t') \overset{(2)}{=} \sum_{t'\in T} p(t)(t') \;=\; 1\end{aligned}$$

The previous computation holds for each state $t$ in $\mathcal{M}'$. It shows that the outgoing probabilities given by $p'$ form a probability distribution for each state in $\mathcal{M}'$ and thus that $\mathcal{M}'$ is an MC. Note that step (1) comes from the definition of $p'$ with respect to $p$ and that step (2) comes from the definition of $T'$ according to $\bar{T}$ and $\bar{t}$.

2. We now prove that $\mathcal{M}'$ satisfies $\mathcal{I}$. $\mathcal{M}$ satisfies $\mathcal{I}$ implies that there a exists a satisfaction relation $\mathcal{R}$ between $\mathcal{M}$ and $\mathcal{I}$. Let $\mathcal{R}' \subseteq T \times S$ be s.t. $t\ \mathcal{R}'\ s$ if $t\ \mathcal{R}\ s$ and $\bar{t}\ \mathcal{R}'\ \bar{s}$ if there exists a state $t' \in \bar{T}$ s.t. $t'\ \mathcal{R}\ \bar{s}$. We prove that $\mathcal{R}'$ is a satisfaction relation between $\mathcal{M}'$ and $\mathcal{I}$. For each pair $(t,s) \in \mathcal{R}$ we note $\delta_{(s,t)}$ the correspondence function given by the satisfaction relation $\mathcal{R}$. Let $(t,s)$ be in $\mathcal{R}'$ and $\delta' : T' \to (S \to [0,1])$ be s.t. $\delta'(t')(s') = \delta_{(t,s)}(t')(s')$ if $t' \neq \bar{t}$ and $\delta'(t')(s') = max_{t'\in\bar{T}}(\delta_{(t,s)}(t')(s'))$ otherwise. $\delta'$ is a correspondence function for the pair $(t,s)$ in $\mathcal{R}'$ such as required by the $\models^a_{\mathtt{I}}$ satisfaction relation:

   a) Let $t'$ be in $T$. If $t' \neq \bar{t}$ then $\delta'(t')$ is equivalent to $\delta_{(t,s)}(t')(s')$ which is by definition a distribution on $S$. Otherwise $t' = \bar{t}$ and the following computation proves that $\delta'(\bar{t})$ is a distribution on $S$. For the step (1) remind that $\mathcal{R}$ is a satisfaction relation with degree 1 and that $\bar{t}\ \mathcal{R}\ \bar{s}$. This implies that $\delta_{(t,s)}(\bar{t})(s')$ equals to zero for all $s' \neq \bar{s}$. For the step (2), $\mathcal{R}$ is a satisfaction relation with degree 1 implies that $\delta_{(t,s)}(t')(s')$ equals to 0 or 1 for all $t' \in T$ and $s' \in S$. Finally the recursive definition of the satisfaction relation $\mathcal{R}$ implies that there exists at least one state $t'' \in \bar{T}$ s.t. $\delta_{(t,s)}(t'')(\bar{s})$ does not equal to zero (*i.e.*, equals to one).

$$\begin{aligned}\sum_{s'\in S} \delta'(\bar{t})(s') &= \sum_{s'\in S\setminus\{\bar{s}\}} \delta'(\bar{t})(s') \;+\; \delta'(\bar{t})(\bar{s}) \\ &= \sum_{s'\in S\setminus\{\bar{s}\}} \delta_{(t,s)}(\bar{t})(s') \;+\; max_{t''\in\bar{T}}(\delta_{(t,s)}(t'')(\bar{s})) \\ &\overset{(1)}{=} max_{t''\in\bar{T}}(\delta_{(t,s)}(t'')(s')) \\ &\overset{(2)}{=} 1\end{aligned}$$

   b) Let $s'$ be in $S$. Step (1) uses the definition of $p'$ according to $p$. Step (2) uses the definition of $\delta'$ according to $\delta_{(t,s)}$. Step (3) comes from the fact that for

all $t, t' \in T \times \bar{T}$, we have by the definition of the satisfaction relation $\mathcal{R}$ with degree 1 and by construction of $\bar{T}$ that if $p(t,t') \neq 0$ then $\delta_{(t)(s)}(t', \bar{s}) = 1$ and $\delta_{(t,s)}(t')(s') = 0$ for all $s' \neq \bar{s}$. Finally, step (4) comes from the definition of the correspondence function $\delta_{(t,s)}$ for the pair $(t,s)$ in $\mathcal{R}$.

$$
\begin{aligned}
&\sum_{t' \in T'} p'(t)(t') \times \delta'(t')(s') \\
=&\sum_{t' \in T' \setminus \{\bar{t}\}} p'(t)(t') \times \delta'(t')(s') \; + \; p'(t,\bar{t}) \times \delta'(\bar{t})(s') \\
\overset{(1)}{=}&\sum_{t' \in T' \setminus \{\bar{t}\}} p(t)(t') \times \delta'(t')(s') \; + \; \sum_{t' \in \bar{T}} p(t)(t') \times \delta'(\bar{t})(s') \\
\overset{(2)}{=}&\sum_{t' \in T' \setminus \{\bar{t}\}} p(t)(t') \times \delta_{(t,s)}(t')(s') \; + \; \sum_{t' \in \bar{T}} p(t)(t') \times max_{t'' \in \bar{T}}(\delta_{(t,s)}(t'')(s')) \\
\overset{(3)}{=}&\sum_{t' \in T' \setminus \{\bar{t}\}} p(t)(t') \times \delta_{(t,s)}(t')(s') \; + \; \sum_{t' \in \bar{T}} p(t)(t') \times \delta_{(t,s)}(t')(s') \\
=&\sum_{t' \in T' \setminus \{\bar{t}\} \cup \bar{T}} p(t)(t') \times \delta_{(t,s)}(t')(s') = \sum_{t' \in T} p(t)(t') \times \delta_{(t,s)}(t')(s') \\
\overset{(4)}{\in}& P(s,s')
\end{aligned}
$$

c) Let $t'$ be in $T'$ and $s'$ be in $S$. We have by construction of $\mathcal{R}'$ from $\mathcal{R}$ that if $\delta'(t')(s') > 0$ then $(t', s') \in \mathcal{R}$.

3. Ne now prove that the probability of reaching $\alpha$ from $\bar{t}$ is lower in $\mathcal{M}'$ than in $\mathcal{M}$. We consider the MC $\mathcal{M}''$ from $\mathcal{M}$ where the states containing the label $\alpha$ are replaced by absorbing states. Formally $\mathcal{M}'' = (T, t_0, p'', V)$ such that for all $t, t' \in T$: $p''(t,t') = p(t,t')$ if $\alpha \not\subseteq V(t)$ else $p''(t,t') = 1$ if $t = t'$ and $p''(t,t') = 0$ otherwise. By definition of the reachability property we get that $\mathbb{P}_t^{\mathcal{M}''}(\Diamond\alpha)$ equals to $\mathbb{P}_t^{\mathcal{M}}(\Diamond\alpha)$ for all state $t$ in $T'$. Following computation concludes the proof. Step (1) comes from Lemma 3. Step (2) comes by construction of $\mathcal{M}'$ from $\mathcal{M}$. Step (3) comes by construction of $\mathcal{M}''$ from $\mathcal{M}$ where states labeled with $\alpha$ are absorbing states. Step (4) comes from the fact that $\mathbb{P}_{t_n}^{\mathcal{M}''}(\Diamond\alpha)$ is equal to $\mathbb{P}_{t_n}^{\mathcal{M}''}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } \alpha) + \Sigma_{1 \leq i \leq n} \mathbb{P}_{t_n}^{\mathcal{M}''}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } t_i) \times \mathbb{P}_{t_i}^{\mathcal{M}''}(\Diamond\alpha)$. Step (5) uses the fact that $\mathbb{P}_{t_i}^{\mathcal{M}}(\Diamond\alpha) \geq \mathbb{P}_{t_n}^{\mathcal{M}}(\Diamond\alpha)$ for all $1 \leq i \leq n$ and by construction this is also correct in $\mathcal{M}''$. Last steps are straightforward.

$$
\begin{aligned}
&\mathbb{P}_{\bar{t}}^{\mathcal{M}'}(\Diamond\alpha) \\
\overset{(1)}{=}&\frac{\mathbb{P}_{\bar{t}}^{\mathcal{M}'}(\neg\bar{t} \text{ U } \alpha)}{1 - \mathbb{P}_{\bar{t}}^{\mathcal{M}'}(\neg\alpha \text{ U } \bar{t})} \\
\overset{(2)}{=}&\frac{\mathbb{P}_{t_n}^{\mathcal{M}}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } \alpha)}{1 - \mathbb{P}_{t_n}^{\mathcal{M}}(\neg\alpha \text{ U } (t_1 \vee \ldots \vee t_n))}
\end{aligned}
$$

$$
\begin{aligned}
&\overset{(3)}{=} \frac{\mathbb{P}^{\mathcal{M}''}_{t_n}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } \alpha)}{1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n))} \\
&\overset{(4)}{=} \frac{\mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha) - \sum_{1 \leq i \leq n} \mathbb{P}^{\mathcal{M}''}_{t_n}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } t_i) \times \mathbb{P}^{\mathcal{M}''}_{t_i}(\Diamond\alpha)}{1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n))} \\
&\overset{(5)}{\leq} \frac{\mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha) - \sum_{1 \leq i \leq n} \mathbb{P}^{\mathcal{M}''}_{t_n}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } t_i) \times \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha)}{1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n))} \\
&\overset{(6)}{=} \frac{\mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha) \times (1 - \sum_{1 \leq i \leq n} \mathbb{P}^{\mathcal{M}''}_{t_n}(\neg(t_1 \vee \ldots \vee t_n) \text{ U } t_i))}{1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n))} \\
&\overset{(7)}{=} \frac{\mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha) \times (1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n)))}{1 - \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond(t_1 \vee \ldots \vee t_n))} \\
&= \mathbb{P}^{\mathcal{M}''}_{t_n}(\Diamond\alpha) \\
&= \mathbb{P}^{\mathcal{M}}_{\bar{t}}(\Diamond\alpha)
\end{aligned}
$$

The same method can be used for proving that $\mathbb{P}^{\mathcal{M}_2}(\Diamond\alpha) \geq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$ by defining $\bar{T} = \{t_1, \ldots, t_n\}$ to be the set $\mathcal{R}^{-1}(s)$ s.t. the states $t_i$ are ordered by *increasing* probability of reaching $\alpha$. Thereby the symbol $\leq$ at step (5) for the computation of $\mathbb{P}^{\mathcal{M}'}_{\bar{t}}(\Diamond\alpha)$ is replaced by the symbol $\geq$. □

Next, Lemma 4 is a consequence of Corollary 1 and Lemma 2 and states that the maximal and the minimal probability of reaching a given proposition is realized by Markov chains with the same structure than the IMC.

**Lemma 4.** *Let $\mathcal{I} = (S, s_0, P, V)$ be an* IMC*, $\mathcal{M}$ be an* MC *satisfying $\mathcal{I}$ w.r.t. $\models^a_{\mathtt{I}}$, and $\alpha \subseteq A$ be a proposition. There exist* MC*s $\mathcal{M}_1$ and $\mathcal{M}_2$ satisfying $\mathcal{I}$ w.r.t. $\models^o_{\mathtt{I}}$ such that $\mathbb{P}^{\mathcal{M}_1}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}_2}(\Diamond\alpha)$.*

*Proof.* Let $\mathcal{I}$ be an IMC and $\mathcal{M}$ be an MC satisfying $\mathcal{I}$ w.r.t. $\models^a_{\mathtt{I}}$. Consider the sequence of MCs $(\mathcal{M}_n)_{n\in\mathbb{N}}$ s.t. $\mathcal{M}_0$ is the MC satisfying $\mathcal{I}$ with degree 1 obtained by Corollary 1 and for all $n \in \mathbb{N}$, $\mathcal{M}_{n+1}$ is the MC satisfying $\mathcal{I}$ with strictly less states than $\mathcal{M}_n$ and verifying $\mathbb{P}^{\mathcal{M}_{n+1}}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}_n}(\Diamond\alpha)$ given by Lemma 2 if $\mathcal{M}_n$ does not have the same structure than $\mathcal{I}$ and equal to $\mathcal{M}_n$ otherwise. By construction $(\mathcal{M}_n)_{n\in\mathbb{N}}$ is finite and its last element is a Markov chain $\mathcal{M}'$ with the same structure than $\mathcal{I}$ s.t. $\mathbb{P}^{\mathcal{M}'}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$. Thus, $\mathcal{M}'$ satisfies $\mathcal{I}$ w.r.t. $\models^o_{\mathtt{I}}$ s.t. $\mathbb{P}^{\mathcal{M}'}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$. The same method can be used for proving the other side of the inequality (*i.e.*, there exists an MC $\mathcal{M}'$ s.t. $\mathcal{M}' \models^o_{\mathtt{I}} \mathcal{I}$ and $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \leq \mathbb{P}^{\mathcal{M}'}(\Diamond\alpha)$). □

Finally, the following proves our Theorem 1 using Lemma 4 and Proposition 3.

*Proof for Theorem 1.* Let $\mathcal{I} = (S, s_0, P, V)$ be an IMC, $\alpha \subseteq A$ be a state label, $\sim \in \{\leq, <, >, \geq\}$ and $0 < p < 1$. Recall that according to an IMC satisfaction relation the property

$\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ holds iff there exists an MC $\mathcal{M}$ satisfying $\mathcal{I}$ (with the chosen semantics) such that $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha)\sim p$.

1. We first prove the equivalence w.r.t. $\models^o_{\mathtt{I}}$ and $\models^a_{\mathtt{I}}$. Recall also that $\models^a_{\mathtt{I}}$ is more general than $\models^o_{\mathtt{I}}$: for all MC $\mathcal{M}$ if $\mathcal{M} \models^o_{\mathtt{I}} \mathcal{I}$ then $\mathcal{M} \models^o_{\mathtt{I}} \mathcal{I}$ (Proposition 3).

   [$\Rightarrow$] Direct from the fact that $\models^a_{\mathtt{I}}$ is more general than $\models^o_{\mathtt{I}}$ (Proposition 3)

   [$\Leftarrow$] $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ with the at-every-step semantics implies that there exists an MC $\mathcal{M}$ s.t. $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}\sim p$. Thus by Lemma 4 we get that there exists an MC $\mathcal{M}'$ s.t. $\mathcal{M}' \models^o_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}'\sim p$.

2. We now prove the equivalence w.r.t. $\models^o_{\mathtt{I}}$ and $\models^d_{\mathtt{I}}$

   [$\Rightarrow$] Direct from the fact that $\models^a_{\mathtt{I}}$ is more general than $\models^o_{\mathtt{I}}$. (Proposition 3)

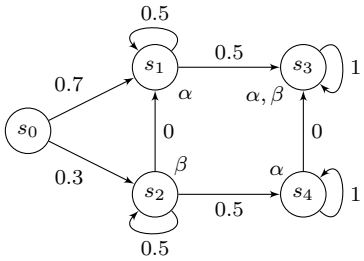
   [$\Leftarrow$] $\mathbb{P}^{\mathcal{I}}(\Diamond\alpha)\sim p$ with the IMDP semantics implies that there exists an MC $\mathcal{M}$ s.t. $\mathcal{M} \models^d_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}\sim p$. Since $\models^a_{\mathtt{I}}$ is more general than $\models^d_{\mathtt{I}}$ we have that $\mathcal{M} \models^a_{\mathtt{I}} \mathcal{I}$. Thus by Lemma 4 we get that there exists an MC $\mathcal{M}'$ s.t. $\mathcal{M}' \models^o_{\mathtt{I}} \mathcal{I}$ and $\mathcal{M}'\sim p$.

□

### 5.5.2 Constraint Encodings

Note that the result from Theorem 1 naturally extends to pIMCs. We therefore exploit this result to construct a CSP encoding for verifying quantitative reachability properties in pIMCs. As in Section 5.4, we extend the CSP $\mathbf{C}_{\exists\mathbf{c}}$, that produces a correct MC implementation for the given pIMC, by imposing that this MC implementation satisfies the given quantitative reachability property. In order to compute the probability of reaching state label $\alpha$ at the MC level, we use standard techniques from [105] that require the partitioning of the state space into three sets $S_\top$, $S_\bot$, and $S_?$ that correspond to states reaching $\alpha$ with probability 1, states from which $\alpha$ cannot be reached, and the remaining states, respectively. Once this partition is chosen, the reachability probabilities of all states in $S_?$ are computed as the unique solution of a linear equation system (see [105], Theorem 10.19, p.766). We now explain how we identify states from $S_\bot, S_\top$ and $S_?$ and how we encode the linear equation system, which leads to the resolution of quantitative reachability.

Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC and $\alpha \subseteq A$ be a state label. We start by setting $S_\top = \{s \mid V(s) = \alpha\}$. We then extend $\mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$ in order to identify the set $S_\bot$. Let $\mathbf{C}'_{\exists\mathbf{r}}(\mathcal{P}, \alpha) = (X \cup X', D \cup D', C \cup C')$ be such that $(X, D, C) = \mathbf{C}_{\exists\mathbf{r}}(\mathcal{P})$, $X'$ contains one Boolean variable $\lambda_s$ and one integer variable $\alpha_s$ with domain $[0, |S|]$ per state $s$ in $S$, $D'$ contains the domains of these variables, and $C'$ is composed of the following constraints for each state $s \in S$:




| state | $\rho_s$ | $\omega_s$ | $\alpha_s$ | $\lambda_s$ |
|---|---|---|---|---|
| $s_0$ | true | 1 | 3 | true |
| $s_1$ | true | 2 | 2 | true |
| $s_2$ | true | 2 | 0 | false |
| $s_3$ | true | 3 | 1 | true |
| $s_4$ | true | 3 | 0 | false |

Figure 5.13: A solution to the CSP $\mathbf{C}'_{\exists\mathbf{r}}(\mathcal{P}, \{\alpha, \beta\})$ for the pIMC $\mathcal{P}$ from Fig. 5.7

**(11)** $\alpha_s = 1$, if $\alpha = V(s)$

**(12)** $\alpha_s \neq 1$, if $\alpha \neq V(s)$

**(13)** $\lambda_s \Leftrightarrow (\rho_s \wedge (\alpha_s \neq 0))$

**(14)** $\alpha_s > 1 \Rightarrow \bigvee_{s' \in \mathtt{Succ}(s)\setminus\{s\}} (\alpha_s = \alpha_{s'} + 1) \wedge (\theta_s^{s'} > 0)$, if $\alpha \neq V(s)$

**(15)** $\alpha_s = 0 \Leftrightarrow \bigwedge_{s' \in \mathtt{Succ}(s)\setminus\{s\}} (\alpha_{s'} = 0) \vee (\theta_s^{s'} = 0)$, if $\alpha \neq V(s)$

Note that variables $\alpha_s$ play a symmetric role to variables $\omega_s$ from $\mathbf{C}_{\exists\mathbf{r}}$: instead of indicating the existence of a path from $s_0$ to $s$, they characterize the existence of a path from $s$ to a state labeled with $\alpha$. In addition, due to Constraint **(13)**, variables $\lambda_s$ are set to true iff there exists a path with non zero probability from the initial state $s_0$ to a state labeled with $\alpha$ passing by $s$. Thus, $\alpha$ cannot be reached from states such that $\lambda_s = \mathsf{false}$. Therefore, $S_\perp = \{s \mid \lambda_s = \mathsf{false}\}$, which is formalised in Proposition 8.

**Proposition 8.** *Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a* pIMC *and $\alpha \subseteq A$ be a state label. There exists an* MC *$\mathcal{M} \models^a_{\mathtt{pI}} \mathcal{P}$ iff there exists a valuation $v$ solution of the* CSP *$\mathbf{C}'_{\exists\mathbf{r}}(\mathcal{P}, \alpha)$ s.t. for each state $s \in S$: $v(\lambda_s)$ is equal to* true *iff $\mathbb{P}^{\mathcal{M}}_s(\Diamond\alpha) \neq 0$.*

**Example 32.** Figure 5.13 presents a solution to the CSP $\mathbf{C}'_{\exists\mathbf{r}}(\mathcal{P}, \{\alpha, \beta\})$ for the pIMC $\mathcal{P}$ from Figure 5.7. First, note that $s_3$ is the only state labelled by $\{\alpha, \beta\}$ in $\mathcal{P}$. By considering the MC $\mathcal{M}$ built from the valuation of the transition variables in Figure 5.13 we have that: $\alpha_0 = 3$, which implies that there exists a path in $\mathcal{M}$ with size 2 reaching $\alpha$ from $s_1$; $\alpha_1 = 2$, which implies that there exists a path in $\mathcal{M}$ with size 1 reaching $\alpha$ from $s_1$; and $\alpha_2 = 0$, which implies that there is no path in $\mathcal{M}$ reaching $\alpha$ from $s_1$, etc. Finally, by Constraint **(13)** we have that: $\lambda_0$, $\lambda_1$, and $\lambda_3$ are true which implies that the states $s_0$, $s_1$, and $s_3$ are reachable in $\mathcal{M}$ and they can reach $\alpha$; $\lambda_2$ and $\lambda_4$ are false which implies that the states $s_2$ and $s_4$ cannot reach $\alpha$ in $\mathcal{M}$.

Finally, we encode the equation system from [105] in a last CSP encoding that extends $\mathbf{C}'_{\exists\mathbf{r}}$. Let $\mathbf{C}_{\exists\overline{\mathbf{r}}}(\mathcal{P}, \alpha) = (X \cup X', D \cup D', C \cup C')$ be such that $(X, D, C) = \mathbf{C}'_{\exists\mathbf{r}}(\mathcal{P}, \alpha)$, $X'$ contains one variable $\pi_s$ per state $s$ in $S$ with domain $[0, 1]$, $D'$ contains the domains of these variables, and $C'$ is composed of the following constraints for each state $s \in S$:

**(16)** $\neg\lambda_s \Rightarrow \pi_s = 0$

**(17)** $\lambda_s \Rightarrow \pi_s = 1$, if $\alpha = V(s)$

**(18)** $\lambda_s \Rightarrow \pi_s = \Sigma_{s' \in \mathtt{Succ}(s)} \pi_{s'} \theta^s_{s'}$, if $\alpha \neq V(s)$

As a consequence, variables $\pi_s$ encode the probability with which state $s$ eventually reaches $\alpha$ when $s$ is reachable from the initial state and 0 otherwise.

**Proposition 9.** *Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a* pIMC *and $\alpha \subseteq A$ be a proposition. There exists an* MC *$\mathcal{M} \models^a_{\mathtt{pI}} \mathcal{P}$ iff there exists a valuation $v$ solution of the* CSP *$\mathbf{C}_{\exists\overline{\mathbf{r}}}(\mathcal{P}, \alpha)$ s.t. $v(\pi_s)$ is equal to $\mathbb{P}^{\mathcal{M}}_s(\Diamond\alpha)$ if $s$ is reachable from the initial state $s_0$ in $\mathcal{M}$ and is equal to 0 otherwise.*

*Proof.* Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC and $\alpha \subseteq A$ be a state label. $\mathbf{C}_{\exists\overline{\mathbf{r}}}$ extends the CSP $\mathbf{C}'_{\exists\mathbf{r}}$ that produces a MC $\mathcal{M}$ satisfying $\mathcal{P}$ (cf. Proposition 8) by computing the probability of reaching $\alpha$ in $\mathcal{M}$. In order to compute this probability, we use standard techniques from [105] that require the partitioning of the state space into three sets $S_\top$, $S_\bot$, and $S_?$ that correspond to states reaching $\alpha$ with probability 1, states from which $\alpha$ cannot be reached, and the remaining states, respectively. Once this partition is chosen, the reachability probabilities of all states in $S_?$ are computed as the unique solution of an equation system (see [105], Theorem 10.19, p.766). Recall that for each state $s \in S$ variable $\alpha_s$ is equal to $\mathsf{true}$ iff $s$ is reachable in $\mathcal{M}$ and $s$ can reach $\alpha$ with a non zero probability. Thus we consider $S_\bot = \{s \mid \alpha_s = \mathsf{false}\}$, $S_\top = \{s \mid V(s) = \alpha\}$, and $S_? = S \setminus (S_\bot \cup S_\top)$. Finally constraints in $\mathbf{C}_{\exists\overline{\mathbf{r}}}$ encodes the equation system from [105] according to chosen $S_\bot$, $S_\top$, and $S_?$. Thus, $\pi_{s_0}$ equals the probability in $\mathcal{M}$ to reach $\alpha$. □

**Example 33** (Example 32 continued)**.** Consider the valuation given in Figure 5.13 as a partial solution to the CSP $\mathbf{C}_{\exists\overline{\mathbf{r}}}(\mathcal{P}, \{\alpha, \beta\})$. Let $\mathcal{M}$ be the MC built from this partial valuation. Since $s_2$ and $s_4$ cannot reach $\{\alpha, \beta\}$ in $\mathcal{M}$ we have that $S_\bot$ contains $s_2$ and $s_4$. Furthermore, $s_3$ is the only state labelled by $\{\alpha, \beta\}$ in $\mathcal{M}$. Thus, $S_\top$ contains $s_3$ and the remaining states $s_0$ and $s_1$ are in $S_?$. Finally, Constraints **(16)**, **(17)**, and **(18)** encode the following system to compute for each state the quantitative reachability of $\{\alpha, \beta\}$ in $\mathcal{M}$:

$$\begin{cases} \pi_0 = 0.7\pi_1 + 0.3\pi_2 \\ \pi_1 = 0.5\pi_1 + 0.5\pi_3 \\ \pi_2 = 0 \\ \pi_3 = 1 \\ \pi_4 = 0 \end{cases} \Leftrightarrow \begin{cases} \pi_0 = 0.7\pi_1 + 0 \\ \pi_1 = 0.5\pi_1 + 0.5 \\ \pi_2 = 0 \\ \pi_3 = 1 \\ \pi_4 = 0 \end{cases} \Leftrightarrow \begin{cases} \pi_0 = 0.7 \\ \pi_1 = 1 \\ \pi_2 = 0 \\ \pi_3 = 1 \\ \pi_4 = 0 \end{cases}$$

Let $p \in [0, 1] \subseteq \mathbb{R}$ be a probability bound. Adding the constraint $\pi_{s_0} \leq p$ (resp. $\pi_{s_0} \geq p$) to the previous $\mathbf{C}_{\exists\overline{\mathbf{r}}}$ encoding allows to determine if there exists a MC $\mathcal{M} \models^a_{\mathtt{pI}} \mathcal{P}$

| Set of benchmarks | #pIMCs | #nodes | #edges | #intervals min | #intervals avg | #intervals max | #paramInBounds min | #paramInBounds avg | #paramInBounds max | #parameters |
|---|---|---|---|---|---|---|---|---|---|---|
| HERMAN N=3 | 27 | 8 | 28 | 0 | 7 | 18 | 0 | 3 | 11 | {2, 5, 10} |
| HERMAN N=5 | 27 | 32 | 244 | 19 | 50 | 87 | 0 | 12 | 38 | {2, 5, 10} |
| HERMAN N=7 | 27 | 128 | 2,188 | 37 | 131 | 236 | 3 | 31 | 74 | {5, 15, 30} |
| EGL L=2; N=2 | 27 | 238 | 253 | 16 | 67 | 134 | 0 | 15 | 57 | {2, 5, 10} |
| EGL L=2; N=4 | 27 | 6,910 | 7,165 | 696 | 1,897 | 3,619 | 55 | 444 | 1,405 | {2, 5, 10} |
| EGL L=4; N=2 | 27 | 494 | 509 | 47 | 136 | 276 | 3 | 32 | 115 | {2, 5, 10} |
| EGL L=4; N=4 | 27 | 15,102 | 15,357 | 1448 | 4,068 | 7,772 | 156 | 951 | 3,048 | {2, 5, 10} |
| BRP M=3; N=16 | 27 | 886 | 1,155 | 16 | 64 | 135 | 1 | 15 | 45 | {2, 5, 10} |
| BRP M=3; N=32 | 27 | 1,766 | 2,307 | 40 | 128 | 275 | 3 | 32 | 129 | {2, 5, 10} |
| BRP M=4; N=16 | 27 | 1,095 | 1,443 | 22 | 80 | 171 | 0 | 20 | 62 | {2, 5, 10} |
| BRP M=4; N=32 | 27 | 2,183 | 2,883 | 49 | 164 | 323 | 3 | 39 | 139 | {2, 5, 10} |
| CROWDS CS=10; TR=3 | 27 | 6,563 | 15,143 | 1,466 | 3,036 | 4,598 | 57 | 235 | 535 | {5, 15, 30} |
| CROWDS CS=5; TR=3 | 27 | 1,198 | 2,038 | 190 | 410 | 652 | 8 | 31 | 76 | {5, 15, 30} |
| NAND K=1; N=10 | 27 | 7,392 | 11,207 | 497 | 980 | 1,416 | 109 | 466 | 1,126 | {50, 100, 250} |
| NAND K=1; N=5 | 27 | 930 | 1371 | 60 | 121 | 183 | 9 | 58 | 159 | {50, 100, 250} |
| NAND K=2; N=10 | 27 | 14,322 | 21,567 | 992 | 1,863 | 2,652 | 197 | 866 | 2,061 | {50, 100, 250} |
| NAND K=2; N=5 | 27 | 1,728 | 2,505 | 114 | 217 | 329 | 23 | 101 | 263 | {50, 100, 250} |

Table 5.1: Benchmarks composed of 459 pIMCs over 5 families used for verifying qualitative properties

such that $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \leq p$ (resp $\geq p$). Formally, let $\sim \in \{\leq, <, \geq, >\}$ be a comparison operator, we write $\not\sim$ for its negation (*e.g.*, $\not\leq$ is $>$). This leads to the following theorem.

**Theorem 2.** *Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a* pIMC*, $\alpha \subseteq A$ be a label, $p \in [0, 1]$, $\sim \in \{\leq, <, \geq, >\}$ be a comparison operator, and $(X, D, C)$ be $\mathbf{C}_{\exists\bar{\mathbf{r}}}(\mathcal{P}, \alpha)$:*

- CSP $(X, D, C \cup (\pi_{s_0} \sim p))$ *is satisfiable iff* $\exists \mathcal{M} \models^{a}_{\mathtt{pI}} \mathcal{P}$ *s.t.* $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \sim p$
- CSP $(X, D, C \cup (\pi_{s_0} \not\sim p))$ *is unsatisfiable iff* $\forall \mathcal{M} \models^{a}_{\mathtt{pI}} \mathcal{P}$*:* $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \sim p$

*Proof.* Let $\mathcal{P} = (S, s_0, P, V, Y)$ be a pIMC, $\alpha \subseteq A$ be a state label, $p \in [0, 1]$, and $\sim \in \{\leq, <, \geq, >\}$ be a comparison operator. Recall that $\mathbf{C}_{\exists\bar{\mathbf{r}}}(\mathcal{P}, \alpha)$ is a CSP s.t. each solution corresponds to an MC $\mathcal{M}$ satisfying $\mathcal{P}$ where $\pi_{s_0}$ is equal to $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha)$. Thus adding the constraint $\pi_{s_0} \sim p$ allows to find an MC $\mathcal{M}$ satisfying $\mathcal{P}$ such that $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \sim p$. This concludes the first item presented in the theorem. For the second item, we use Theorem 1 with Proposition 9 which ensure that if the CSP $\mathbf{C}_{\exists\bar{\mathbf{r}}}(\mathcal{P}, \alpha)$ to which is added the constraint $\pi_{s_0} \not\sim p$ is not satisfiable then there is no MC satisfying $\mathcal{P}$ w.r.t. $\models^{a}_{\mathtt{pI}}$ such that $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \not\sim p$; thus $\mathbb{P}^{\mathcal{M}}(\Diamond\alpha) \sim p$ for all MC satisfying $\mathcal{P}$ w.r.t. $\models^{a}_{\mathtt{pI}}$. □

## 5.6 Prototype Implementation and Experiments

Our results have been implemented in a prototype tool[5] which generates the above CSP encodings, and CSP encodings from [103] as well. In this section, we first present our

[5]All resources, benchmarks, and source code are available online as a Python library at https://github.com/anicet-bart/pimc_pylib

```
// nand multiplex system
// gxn/dxp 20/03/03
// U (correctly) performs a random permutation of the outputs of the previous stage
dtmc

const int N = 3; // number of inputs in each bundle
const int K = 1; // number of restorative stages

const int M = 2*K+1; // total number of multiplexing units

// parameters originally taken from the following paper and edited by Bart et Al.
// A system architecture solution for unreliable nanoelectric devices
// J. Han & P. Jonker (IEEEE trans. on nanotechnology vol 1(4) 2002)
const double p1; // probability 1st nand works correctly
const double p2; // probability 2nd nand works correctly
const double p3; // probability 3nd nand works correctly
const double prob1; // probability initial inputs are stimulated

// model whole system as a single module by resuing variables to decrease the state space
module multiplex
  u  : [1..M]; // number of stages
  c  : [0..N]; // counter (number of copies of the nand done)

  s  : [0..4]; // local state
  // 0 - initial state ; 1 - set x inputs ; 2 - set y inputs
  // 3 - set outputs   ; 4 - done

  z  : [0..N]; // number of new outputs equal to 1
  zx : [0..N]; // number of old outputs equal to 1
  zy : [0..N]; // need second copy for y

  x  : [0..1]; // value of first input
  y  : [0..1]; // value of second input

  [] s=0 & (c<N) -> (s'=1); // do next nand if have not done N yet
  [] s=0 & (c=N) & (u<M) -> (s'=1)&(zx'=z)&(zy'=z)&(z'=0)&(u'=u+1)&(c'=0); // next gate
  [] s=0 & (c=N) & (u=M) -> (s'=4)&(zx'=0)&(zy'=0)&(x'=0)&(y'=0); // finished

  // choose x permute selection (have zx stimulated inputs)
  // note only need y to be random
  [] s=1 & u=1  -> prob1: (x'=1)&(s'=2) + (1-prob1): (x'=0)&(s'=2); // initially random
  [] s=1 & u>1 & zx>0 -> (x'=1) & (s'=2) & (zx'=zx-1);
  [] s=1 & u>1 & zx=0 -> (x'=0) & (s'=2);

  // choose x randomly from selection (have zy stimulated inputs)
  [] s=2 & u=1 -> prob1: (y'=1)&(s'=3) + (1-prob1): (y'=0)&(s'=3); // initially random
  [] s=2 & u>1 & zy<(N-c) & zy>0  ->
    zy/(N-c): (y'=1)&(s'=3)&(zy'=zy-1) + 1-(zy/(N-c)): (y'=0)&(s'=3);
  [] s=2 & u>1 & zy=(N-c) & c<N -> 1: (y'=1) & (s'=3) & (zy'=zy-1);
  [] s=2 & u>1 & zy=0 -> 1: (y'=0) & (s'=3);

  // use 1st nand gate with p1 as error probability
  [] s=3 & z<N & c=0 ->
    (1-p1): (z'=z+(1-x*y)) & (s'=0) & (c'=c+1) & (x'=0) & (y'=0)  // not faulty
    + p1  : (z'=z+(x*y))   & (s'=0) & (c'=c+1) & (x'=0) & (y'=0); // von neumann fault
  // use 2nd nand gate with p2 as error probability
  [] s=3 & z<N & c=1 ->
    (1-p2): (z'=z+(1-x*y)) & (s'=0) & (c'=c+1) & (x'=0) & (y'=0)  // not faulty
    + p2  : (z'=z+(x*y))   & (s'=0) & (c'=c+1) & (x'=0) & (y'=0); // von neumann fault
  // use 3rd nand gate with p3 as error probability
  [] s=3 & z<N & c=2 ->
    (1-p3): (z'=z+(1-x*y)) & (s'=0) & (c'=c+1) & (x'=0) & (y'=0)  // not faulty
    + p3  : (z'=z+(x*y))   & (s'=0) & (c'=c+1) & (x'=0) & (y'=0); // von neumann fault

  [] s=4 -> true;
endmodule

// rewards: final value of gate
rewards
  [] s=0 & (c=N) & (u=M) : z/N;
endrewards
```

Figure 5.14: NAND K=1; N=3 benchmark formulated in the PRISM adapted from [110].

benchmark, then we evaluate our tool for the qualitative properties, and we conclude with the quantitative properties.

### 5.6.1 Benchmark

MCs have been used for many decades to model real-life applications. PRISM[110] is a reference for the verification of probabilistic systems. In particular, it is able to verify properties for MCs. As said in Section 5.2, pIMCs correspond to abstractions of MCs. PRISM references several benchmarks based on MCs[6]. Note first that we only consider pIMCs with linear parametric expressions. In this context all the CSPs encodings for verifying the qualitative properties only use linear constraints while the CSPs encodings for verifying the quantitative properties produce quadratic constraints (*i.e.*, non-linear constraints). This produces an order of magnitude between the time complexity for solving the qualitative properties vs the quantitative properties w.r.t. our encodings. Thus, we consider two different benchmarks presented in Table 5.1 and 5.2. In both cases, pIMCs are automatically generated from the PRISM model in a text format inspired from [107].

For the first benchmark used for verifying qualitative properties, we constructed the pIMCs from existing MCs by randomly replacing some exact probabilities on transitions by (parametric) intervals of probabilities. Our pIMC generator takes 4 arguments: the MC transition function; the number of parameters for the generated pIMC; the ratio of the number of intervals over the number of transitions in the generated pIMC; the ratio of the number of parameters over the number of interval endpoints for the generated pIMC. The benchmarks used are presented in Table 5.1. We selected 5 applications from PRISM [110]: HERMAN - the self-stabilisation protocol of Herman from [111]; EGL - the contract signing protocol of Even, Goldreich & Lempel from [112]; BRP - the bounded retransmission protocol from [113]; CROWDS - the crowds protocol from [114]; and NAND - the nand multiplexing from [115]. Each one is instantiated by setting global constants (*e.g.*, N for the application HERMAN, L and N for the application EGL) leading to more or less complex MCs. We used our pIMC generator to generate an heterogeneous set of benchmarks: 459 pIMCs with 8 to $15,102$ states and 28 to $21,567$ transitions not reduced to $[0,0]$. The pIMCs contain from 2 to 250 parameters over 0 to 7772 intervals.

For the second benchmark used for verifying quantitative properties we extended the NAND model from [115]. The original MC NAND model has already been extended as a pMC in [98], where the authors consider a single parameter $p$ for the probability that each of the $N$ *nand* gates fails during the multiplexing. We extend this model to pIMC by considering one parameter for the probability that the initial inputs are stimulated and we have one parameter per *nand* gate to represent the probability that it fails. We consider 4 pIMCs with 104 to $7,392$ states and 147 to $11,207$ transitions not reduced to $[0,0]$. The pIMCs contain from 4 to 12 parameters appearing over 82 to $5,698$ transitions.

[6] see the category discrete-time Markov chains on the PRISM website

| Benchmarks | #nodes | #edges | #paramInBounds | #parameters |
|---|---|---|---|---|
| NAND K=1; N=2 | 104 | 147 | 82 | 4 |
| NAND K=1; N=3 | 252 | 364 | 200 | 5 |
| NAND K=1; N=5 | 930 | 1,371 | 726 | 7 |
| NAND K=1; N=10 | 7,392 | 11,207 | 5698 | 12 |

Table 5.2: Benchmarks composed of 4 pIMCs used for verifying quantitative properties

| Encoding | Size of the produced CSPs | Boolean var. | Integer var. | Real-number var. | Boolean constr. | Linear constr. | Quadratic constr. |
|---|---|---|---|---|---|---|---|
| **SotA** | exponential | no | no | yes | yes | yes | no |
| $\mathbf{C}_{\exists\mathbf{c}}$ | linear | yes | no | yes | yes | yes | no |
| $\mathbf{C}_{\exists\mathbf{r}}$ | linear | yes | yes | yes | yes | yes | no |
| $\mathbf{C}_{\exists\overline{\mathbf{r}}}$ | linear | yes | yes | yes | yes | yes | yes |

Table 5.3: Characteristics of the four CSP encodings **SotA**, $\mathbf{C}_{\exists\mathbf{c}}$, $\mathbf{C}_{\exists\mathbf{r}}$, and $\mathbf{C}_{\exists\overline{\mathbf{r}}}$.

### 5.6.2 Constraint Modelling

Given a pIMC in a text format our tool produces the desired CSP according to the selected encoding (*i.e.*, one from [103], $\mathbf{C}_{\exists\mathbf{c}}$, $\mathbf{C}_{\exists\mathbf{r}}$, or $\mathbf{C}_{\exists\overline{\mathbf{r}}}$). Recall that our benchmark only consider linear parametric expressions on transitions. The choice of the contraint programming language for implementing a CSP encoding depends on its nature (*e.g.*, the type of the variables: integer vs. continuous, the kind of the contraints: linear vs. non-linear). Table 5.3 summarizes the natures the encodings where **SotA** stands the encoding from [103] answering the existential consistency problem. Thus, **SotA**, $\mathbf{C}_{\exists\mathbf{c}}$, and $\mathbf{C}_{\exists\mathbf{r}}$ can be implemented as Mixed Integer Linear Programs (`MILP`) [116] and as Satisfiability Modulo Theory (`SMT`) programs [117] with QF_LRA logic (Quantifier Free linear Real-number Arithmetic). This logic deals with Boolean combinations of inequations between linear polynomials over real variables. Note that, QF_NRA does not deal with integer variables. Indeed logics mixing integers and reals are harder than those over reals only. However, all the integer variables in our encodings can be replaced by real-number variables.[7] Each integer variable $x$ can be declared as a real variable whose real domain bounds are its original integer domain bounds; we also add the constraint $x < 1 \Rightarrow x = 0$. Since we only perform incrementation of $x$ this preserves the same set of solutions (*i.e.*, ensures integer integrity constraints). Finally, due to the non-linear constraints in $\mathbf{C}_{\exists\overline{\mathbf{r}}}$, these encodings are implemented as `SMT` programs [117] with the QF_NRA logic (Quantifier Free Non linear Real-number Arithmetic). We use the same technique than for $\mathbf{C}_{\exists\mathbf{c}}$ and $\mathbf{C}_{\exists\mathbf{r}}$ for replacing integer variables by real-number variables. We chose the programming language Python for implementing our CSP modeller. We do not evaluate any arithmetic expression while generating CSPs, and numbers in the interval endpoints of the pIMCs are read as strings and no trivial simplification is performed while modelling. We do so

[7] Note that this is not always free to obtain integer integrity constraints over real-numbers.

| Set of benchmarks | | #variables (1) | (2) | (3) | #constraints (1) | (2) | (3) | avg(model. time) (1) | (2) | (3) | avg(solv. time) (1) | (2) | (3) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HERMAN | N=3 | 71 | 42 | 42 | 1258 | 272 | 238 | 0.10 | 0.07 | 0.07 | 0.01 | 0.01 | 0.01 |
| HERMAN | N=5 | 1,031 | 282 | 282 | 51,064 | 2,000 | 1,750 | 1.52 | 0.08 | 0.08 | 0.24 | 0.03 | 0.01 |
| HERMAN | N=7 | 16,402 | 2,333 | 2,333 | 1,293,907 | 16,483 | 14,278 | 50.47 | 0.13 | 0.13 | 5.92 | 0.28 | 0.04 |
| EGL | L=2; N=2 | 462 | 497 | 497 | 4,609 | 3,917 | 3,658 | 0.21 | 0.08 | 0.08 | 0.02 | 0.04 | 0.01 |
| EGL | L=2; N=4 | 13,786 | 14,081 | 14,081 | 138,596 | 112,349 | 105,178 | 5.66 | 0.44 | 0.44 | 0.54 | 2.15 | 0.36 |
| EGL | L=4; N=2 | 958 | 1,009 | 1,009 | 9,560 | 8,013 | 7,498 | 0.36 | 0.10 | 0.10 | 0.04 | 0.08 | 0.02 |
| EGL | L=4; N=4 | 30,138 | 30,465 | 30,465 | 301,866 | 243,421 | 228,058 | 13.03 | 0.87 | 0.87 | 1.26 | 11.31 | 0.97 |
| BRP | MAX=3; N=16 | 68,995 | 2,047 | 2,047 | 738,580 | 16,063 | 14,902 | 32.29 | 0.12 | 0.12 | 3.54 | 0.21 | 0.06 |
| BRP | MAX=3; N=32 | OM | 4,079 | 4,079 | OM | 32,047 | 29,734 | OM | 0.18 | 0.18 | OM | 0.47 | 0.13 |
| BRP | MAX=4; N=16 | 103,105 | 2,544 | 2,544 | 1,114,774 | 19,960 | 18,511 | 46.54 | 0.13 | 0.13 | 5.42 | 0.27 | 0.08 |
| BRP | MAX=4; N=32 | OM | 5,072 | 5,072 | OM | 39,832 | 36,943 | OM | 0.21 | 0.21 | OM | 0.63 | 0.17 |
| CROWDS | CS=10; TR=3 | OM | 21,723 | 21,723 | OM | 165,083 | 149,923 | OM | 0.67 | 0.66 | OM | 11.48 | 0.79 |
| CROWDS | CS=5; TR=3 | OM | 3,253 | 3,253 | OM | 25,063 | 23,008 | OM | 0.16 | 0.15 | OM | 0.39 | 0.09 |
| NAND | K=1; N=10 | 87,506 | 18,732 | 18,732 | 888,733 | 145,108 | 133,768 | 152.06 | 0.56 | 0.56 | 3.72 | 6.21 | 0.79 |
| NAND | K=1; N=5 | 6,277 | 2,434 | 2,434 | 62,987 | 18,098 | 16,594 | 10.26 | 0.12 | 0.12 | 0.24 | 0.25 | 0.07 |
| NAND | K=2; N=10 | 169,786 | 36,022 | 36,022 | 1,722,970 | 279,998 | 258,298 | 298.93 | 1.04 | 1.04 | 7.75 | 31.81 | 2.06 |
| NAND | K=2; N=5 | 11,623 | 4,366 | 4,366 | 117,814 | 33,218 | 30,580 | 19.24 | 0.17 | 0.17 | 0.44 | 0.48 | 0.13 |

Table 5.4: Comparison of sizes, modelling, and solving times for three approaches: (1) **SotA** encoding implemented in SMT, (2) $\mathbf{C}_{\exists\mathbf{c}}$ encoding implemented in SMT, and (3) $\mathbf{C}_{\exists\mathbf{c}}$ encoding implemented in MILP (times are given in seconds).

to avoid any rounding of the interval endpoints when using floating point numbers.

Experiments have been realized on a 2.4 GHz Intel Core i5 processor. Time out has been set to 10 minutes. Memory out has been set to 2Gb. Table 5.4 presents the size of the instances (*i.e.*, the number of variables and the number of constraints) for solving the existential consistency problem on our benchmark using (1) `SMT` **SotA** encoding, (2) `SMT` $\mathbf{C}_{\exists\mathbf{c}}$ encoding, and (3) `MILP` $\mathbf{C}_{\exists\mathbf{c}}$ encoding. First, note that all the pIMCs are successfully compiled when using our $\mathbf{C}_{\exists\mathbf{c}}$ encoding while the **SotA** encoding produces out of memory errors for 4 sets of benchmarks: more than 20% of the instances (see OM cells in Table 5.4). We recall that the **SotA** encoding is defined inductively and that it iterates over the power set of the states. In practice, this implies deep recursions joined with enumeration over the power set of the states. The exponential gain exposed in Section 5.4 is visible in terms of number of variables and constraints in Table 5.4, and in terms of encoding time in Figure 5.15. Each dot in Figure 5.15 corresponds to one instance of our benchmark. While the encoding time ranges between 0 and 1s when using the $\mathbf{C}_{\exists\mathbf{c}}$ encoding, it varies between 0 and 500s when using the **SotA** encoding (if it does not run out of memory).

`MILP` formulation of logical constraints (*e.g.*, conjunction, disjunction, implication, equivalence) implies the introduction of binary variables called indicator variables [118]. Each indicator variable is associated to one or more constraints. The valuation of the indicator variable activates or deactivates its associated constraints. We tried to formulate the **SotA** encoding into `MILP`. Unfortunately, the nested conjunctions and disjunctions imply the introduction of a huge number of indicator variables, leading to giant instances giving bad encoding and solving time. However, since the Boolean variables in $\mathbf{C}_{\exists\mathbf{c}}$ exactly correspond to indicator variables, the `MILP` formulation of the $\mathbf{C}_{\exists\mathbf{c}}$ encoding does not

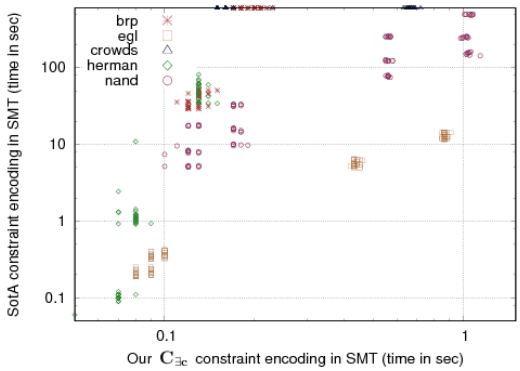



Figure 5.15: Comparing encoding time for the existential consistency problem

introduce additional variables or constraints. The difference between $\mathbf{C}_{\exists \mathbf{c}}$ in SMT and $\mathbf{C}_{\exists \mathbf{c}}$ in MILP comes from the encoding of the domains of the continuous variables: in SMT, it requires the use of inequality constraints, *e.g.*, $0 \leq x \leq 1$. The encoding time is the same for SMT and MILP $\mathbf{C}_{\exists \mathbf{c}}$ encoding.

### 5.6.3 Solving

We chose Z3 [119] in its last version (v. 4.4.2) as SMT solver. We chose CPLEX [11] in its last version (v. 12.6.3.0) as MILP solver. Both solvers have not been tuned and we use their default strategies. Experiments have been realized on a 2.4 GHz Intel Core i5 processor. Time out has been set to 10 minutes.

Table 5.4 presents the resolution time for the existential consistency problem on our first benchmark using (1) SMT **SotA** encoding, (2) SMT $\mathbf{C}_{\exists \mathbf{c}}$ encoding, and (3) MILP $\mathbf{C}_{\exists \mathbf{c}}$ encoding. While the **SotA** CSPs are larger than the $\mathbf{C}_{\exists \mathbf{c}}$ CSPs, the solving time for the **SotA** CSPs appears to be competitive compared to the solving time for the $\mathbf{C}_{\exists \mathbf{c}}$ CSPs. The scatter plot in Figure 5.16 (logarithmic scale) compares solving times for the SMT **SotA** encoding and SMT $\mathbf{C}_{\exists \mathbf{c}}$ encoding. However when considering the resolution time of the problem (*i.e.*, the encoding time plus the solving time) the $\mathbf{C}_{\exists \mathbf{c}}$ encoding clearly answers faster than the **SotA** encoding. Finally, the comparison between the solving time using SMT $\mathbf{C}_{\exists \mathbf{c}}$ encoding and MILP $\mathbf{C}_{\exists \mathbf{c}}$ encoding is illustrated in Figure 5.17. It shows that the loss of safety by passing from real numbers with Z3 SMT resolution to floating point numbers with CPLEX MILP resolution leads to a non negligible gain in terms of resolution time (near to an exponential gain in our benchmark). Indeed the SMT $\mathbf{C}_{\exists \mathbf{c}}$ encoding requires 50 seconds to complete the solving process while the MILP $\mathbf{C}_{\exists \mathbf{c}}$ encoding needs less than 5 seconds for the same instances.

Table 5.5 summarizes the results w.r.t. our second benchmark: the pIMC sizes (in terms of states, transitions, and parameters), the CSP sizes (in terms of number of variables and constraints), and the resolution time using the Z3 solver. Note first that we perform pre-processing when verifying reachability properties: *i.e.*, we eliminate all the

| Benchmark | pIMC | | | $\mathbf{C}_{\exists\mathbf{c}}$ | | | $\mathbf{C}_{\exists\mathbf{r}}$ | | | $\mathbf{C}_{\exists\bar{\mathbf{r}}}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #states | #trans. | #par. | #var. | #cstr. | time | #var. | #cstr. | time | #var. | #cstr. | time |
| NAND K=1; N=2 | 104 | 147 | 4 | 255 | 1,526 | 0.17s | 170 | 1,497 | 0.19s | 296 | 2,457 | 69.57s |
| NAND K=1; N=3 | 252 | 364 | 5 | 621 | 3,727 | 0.24s | 406 | 3,557 | 0.30s | 703 | 5,828 | 31.69s |
| NAND K=1; N=5 | 930 | 1,371 | 7 | 2,308 | 13,859 | 0.57s | 1,378 | 12,305 | 0.51s | 2,404 | 20,165 | T.O. |
| NAND K=1; N=10 | 7,392 | 11,207 | 12 | 18,611 | 111,366 | 9.46s | 9,978 | 89,705 | 13.44s | 17,454 | 147,015 | T.O. |

Table 5.5: Comparison of solving times between qualitative and quantitative encodings.

states that cannot reach the goal states. This explains why $\mathbf{C}_{\exists\mathbf{r}}$ has less variables and constraints than $\mathbf{C}_{\exists\mathbf{c}}$. Finally, note the order of magnitude between the resolution time required for solving the qualitative properties vs the quantitative properties w.r.t. our encodings. Indeed, we did not succeed in solving pIMCs with more than 300 states and 400 transitions for quantitative properties while we verified pIMCs with more than 10,000 states and 20,000 transitions in the qualitative context.

## 5.7 Conclusion and Perspectives

In this chapter, we have compared several Markov Chain abstractions in terms of succinctness and we have shown that Parametric Interval Markov Chain is a strictly more succinct abstraction formalism than other existing formalisms such as Parametric Markov Chains and Interval Markov Chains. In addition, we have proposed constraint encodings for checking several properties over pIMC. In the context of qualitative properties such as existential consistency or consistent reachability, the size of our encodings is significantly smaller than other existing solutions. In the quantitative setting, we have compared the three semantics for IMCs and pIMCs and showed that the semantics are equivalent with respect to quantitative reachability properties. As a side effect, this result ensures that all existing tools and algorithms solving reachability problems in IMCs under the once-and-for-all semantics can safely be extended to the at-every-step semantics with no

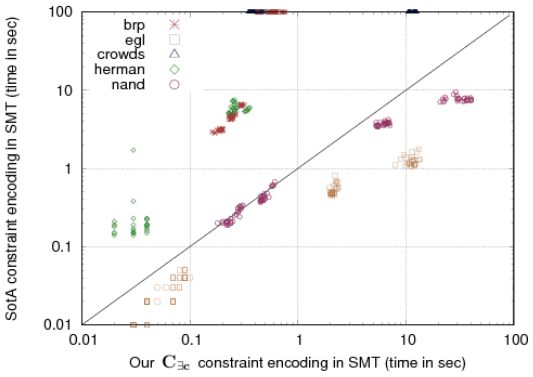



Figure 5.16: Comparing solving time for the existential consistency problem

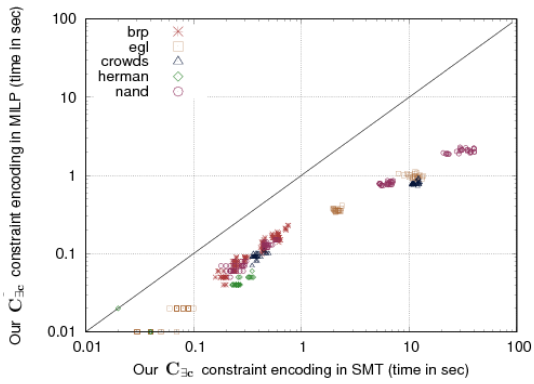



Figure 5.17: Comparing solving time between `SMT` and `MILP` formulations

changes. Based on this result, we have then proposed CSP encodings addressing quantitative reachability in the context of pIMCs regardless of the chosen semantics. Finally, we have developed a prototype tool that automatically generates our CSP encodings and that can be plugged to any constraint solver accepting the SMT-LIB format as input.

Our tool for pIMC verification could be extended in order to manage other, more complex, properties (*e.g.*, supporting the LTL-language in the spirit of what Tulip [107] does). Also one could investigate a practical way of computing and representing the set of *all solutions* to the parameter synthesis problem.

# Chapter 6

# Conclusion and Perspectives

In this thesis we tackled two families of program verification problems. In both cases we first investigated the nature of the verification problem in order to propose an accurate constraint resolution. Since we had no a priori restrictions on the constraint language, we proposed constraint models using non-linear constraints with unbounded continuous variables, mixed integer/continuous domain variables over linear constraints, and also quadratic constraints over mixed variables. We first present in this chapter the conclusions and perspectives of both contributions. We close this thesis with a general conclusion on the benefits of considering constraint programming for program verification.

**Block-Diagram Verification** Block-diagrams are used to model real-time systems such as digital signal processes. Such systems appear in many applications receiving and processing digital signals: modems, multimedia devices, GPS, audio and video processing. We proposed a constraint model using our global constraint called `real-time-loop` for computing over-approximations of real-time streams, based on their block-diagrams representations. We introduced a global constraint and presented a dedicated filtering algorithm inspired by Abstract Interpretation. The experiments show that our approach can reach very good over-approximations in a short running time. Thus, our proposal has been taken in consideration for a future implementation into the FAUST compiler. More generally, our method shows that constraint programming can handle block-diagram analyses in an elegant and natural way.

However we point out some perspectives. Firstly, the propagation loop may be improved according to the tackled verification problem (for instance, when verifying output streams one should favor input to output propagations instead of an arbitrary scheme). Secondly, some constraint patterns offer poor over-approximations when considering interval extensions (*e.g.*, the interval extension of the pattern $|x - int(x)|$ which should compute the decimal part of $x$ returns the interval abstraction $[0; 20]$ with $[-10; 10]$ as input domain whereas the concrete output domain of this pattern is $[0; 1]$). Thus, managing such patterns would improve the quality of the computed over-approximations. Thirdly,

one should test our approach on a language which is not dedicated to audio processing in order to test the practical robustness of our approach with respect to the nature of the programs. Finally, over-approximations intervals ensure that there are no stream values outside the intervals but cannot conclude if they contain at least one stream value. Thus, one may investigate inner-approximations in order to certify the presence of stream values. This could be used to partition the space in three: the intervals that contain only stream values, the intervals that may contain stream values, and the intervals that do not contain any stream value.

**Markov Chain Abstraction Verification** Markov chains model softwares and real-life systems for which probabilities play a fundamental role. We considered the Parametric Interval Markov Chain (pIMC for short) specification formalism for abstracting Markov chains. We first presented a formal theorem proving the equivalence of the three main pIMC semantics with respect to the reachability property. Then, we exploited this result for proposing constraint modellings answering consistency, qualitative, and quantitative reachability properties. For consistency and qualitative reachability, the state-of-the-art constraint models had an exponential size in terms of the verified pIMCs. We proposed constraints models with a linear size in terms of the pIMC size for solving the same problems using the same type of constraints and variables. For quantitative reachability, there was no existing verification process. We thus proposed the first verification process as constraint models in order to answer this problem. Furthermore, we took benefits of the constraint programming paradigm to propose modular constraints models: *i.e.*, the quantitative models extend the qualitative models which extend the consistency models. We implemented our constraint models and we evaluated our prototype over a pIMC benchmark generated from PRISM programs. Constraint models have been generated as mixed linear integer programs and satisfiability modulo theory programs and we obtained promising results. In practice, these results lead to pIMCs closer to the effective resolution of real-word problems.

We now present some perspectives. Firstly, parameters in pIMCs may correspond to possibly controllable inputs in the probabilistic systems or may model a cost to minimize or maximize. Thus, by adding an optimization function to our constraint encodings one may investigate such problems. Secondly, parameters may correspond to decisions to be taken for implementing the pIMC as an IMC in the real-world. The visualization of the parameters state space according to the satisfiability with respect to the property to verify helps to select accurate parameter valuations. However, while some research has been realized with 2 or 3 parameters, one should also investigate cases with more parameters. Finally, the pIMC specification formalism allows to abstract sets of Markov chains. It appears that our constraint encoding may offer another specification formalism. Indeed, one should take benefit of our constraint modellings for expressing guards, relations between

parameters, constraints over some probabilities on transitions, etc. Thus, all the expresivness of the constraint tools could be used for modelling and solving the verification of "constrained Markov chains".

To conclude, program verification is such a rich domain that it can potentially use many constraint tools. The theoretical complexity of program verification such as constraint satisfaction may belong to high complexity levels. Nevertheless, constraint programming solvers may offer a practical resolution to some hard problems However, several constraint programming communities chose different directions and they develop solvers dedicated to separate constraint languages. In this thesis, we considered verification problems through the prism of constraint programming. We proposed constraint programming approaches using various constraint languages for solving the considered verification problems. Our contributions help both fields of constraint programming and program verification to move closer together.

# French summary

## Introduction

La programmation par contraintes est un champ de recherche rattaché à l'intelligence artificielle. Un des objectifs de l'intelligence artificielle est de proposer des méthodes et des outils permettant de réaliser des tâches considérées comme complexes tant à un niveau logique, qu'à un niveau algorithmique. Ainsi le Graal de l'intelligence artificielle est de trouver une solution, un outil capable de résoudre une variété la plus grande possible de problèmes hétérogènes. C'est avec cet objectif que la programmation par contraintes se propose de résoudre tout un ensemble de problèmes qui peuvent être formulés à partir de contraintes. Une contrainte est une relation posée sur un ensemble de variables restreignant les affectations possibles entre les variables et leurs valeurs. En effet, une variable est un objet mathématique associé à un ensemble de valeurs pouvant lui être affectées. Ainsi, nous appelons valuation le choix de valeurs pour les variables, et satisfaire une contrainte revient à trouver une valuation qui satisfasse toutes les relations variables/valeurs établies par les contraintes. La modélisation en contraintes regroupe les différentes techniques utilisées pour passer d'un problème présenté en langage naturel vers un problème formellement décrit mathématiquement ou sous la forme d'un programme en contraintes (LP format, DIMACS format, XCSP format) appelé *modèle*. Une fois l'étape de modélisation terminée, le modèle est envoyé dans le système intelligent, appelé *solver*, pour être résolu. La première étape est appelée modélisation (en : *modelling*) et la seconde résolution (en : *solving*). Dans cette thèse, nous nous intéressons à la modélisation et à la résolution d'applications ciblées : la vérification de programmes.

Lors de ces dernières décennies, l'informatique s'est démocratisée tant dans les usages privés que professionnels. Ainsi, ordinateurs et systèmes d'informations réalisent des applications les plus variées : application intelligentes, systèmes embarqués d'avions, robots médicaux, etc. Comme c'est le cas dans le cadre des chaînes de production, l'écriture de ces systèmes/applications doit respecter un certain nombre de règles de qualité telles que la conformité, l'efficacité, la robustesse. La vérification de programmes a pour objectif

de s'assurer qu'une application, un programme, un système réponde aux spécifications données, c'est-à-dire que son comportement soit correct, qu'il ne contienne pas de "bug". En effet, l'histoire a démontré la nécessité de la mise en oeuvre de telles vérifications. Qu'il s'agisse de la fusée Ariane 5 qui a explosé 36 secondes après son décalage ou du défaut de l'unité de calcul du Pentium II d'Intel qui a causé une perte de 475 millions de dollars et qui a nui gravement à l'image de la marque, ces deux évènements auraient pu être évités s'ils avaient été certifiés, vérifiés formellement d'un point de vue logiciel/programme informatique. Pour autant, la vérification de programme est une tâche difficile car c'est un problème indécidable : en général, il n'est pas possible de construire un système capable de déterminer en temps fini si un programme est correct ou non. Pour autant, indécidable ne veut pas dire infaisable en pratique. Dans cette thèse, nous modélisons et résolvons via la programmation par contraintes des problèmes de vérification de programmes. Nous présentons dans les deux sections suivantes un résumé des deux chapitres contributions de la thèse. Chacun porte sur un problème de vérification de programmes et propose une résolution en contraintes.

# Vérification en contraintes d'un langage temps réel

La programmation par contraintes s'attaque en général à des problèmes statiques, sans notion de temps. Cependant, les méthodes de réduction de domaines pourraient par exemple être utiles dans des problèmes portant sur des flux. C'est le cas de la vérification de programmes temps réel où les variables peuvent changer de valeur à chaque pas de temps. Pour cette contribution, nous nous intéressons à la vérification de domaines de variables (flux) dans le cadre d'un langage de diagrammes de blocs. La première contribution de cette thèse (Chapitre 4) propose une méthode de réduction de domaines de ces flux, pour encadrer finement les valeurs prises au cours du temps. En particulier, nous proposons une nouvelle contrainte globale `real-time-loop`, nous présentons une application au langage FAUST (un langage fonctionnel temps réel pour le traitement audio) et nous testons notre approche sur différents programmes FAUST.

## Contexte et problématique

Comme précisé en introduction de cette section, nous souhaitons vérifier un langage temps réel. Plus précisément, ce langage se positionne dans la famille des diagrammes de blocs que nous présentons ci-dessous. Nous terminons cette section par présenter la problématique de vérification traitée dans cette contribution.

Un *bloc* est une fonction appliquant un opérateur à un ensemble d'entrées ordonnées et produisant une ou plusieurs sorties ordonnées. A partir de là, un *connecteur* relie une sortie d'un bloc à une entrée d'un bloc. Nous appelons un *diagramme de blocs* un ensemble de blocs reliés par des connecteurs. Formellement, notons $E$ un ensemble non vide. Un

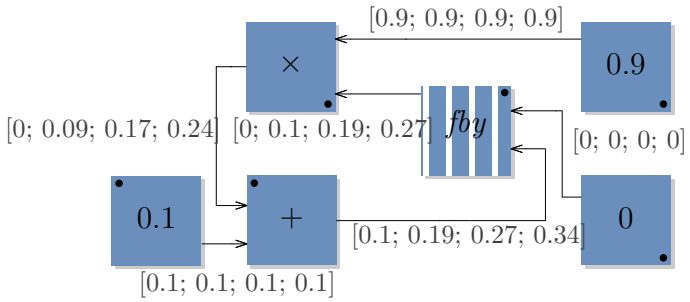



Figure 1: A block-diagram over streams from $\mathtt{BD}(\mathbb{S}(\mathbb{R}))$. In brackets, the first values of the model for $t = 0, 1, 2, 3$.

*bloc* est un triplet $(\mathtt{op}, n, m)$ dont $n \in \mathbb{N}$ est appelé le nombre d'entrées du bloc, $m \in \mathbb{N}$ est appelé le nombre de sorties du bloc et $\mathtt{op} : E^n \to E^m$ est appelé l'opérateur du bloc. De plus, les $n$ entrées (resp. les $m$ sorties) sont ordonnées et $[i]b$ (resp. $b[i]$) correspond à la $i^{\text{ème}}$ entrée (resp. $i^{\text{ème}}$ sortie) avec $1 \leq i \leq n$ (resp. $1 \leq i \leq m$). Un *connecteur* (défini sur un ensemble de blocs $B$) est un couple $(b[i], [j]b')$ appariant les blocs $b$ et $b'$ de $B$ dont la sortie $i$ du bloc $b$ et l'entrée $j$ du bloc $b'$ existent. Ainsi, un *diagramme de blocs* (défini sur $E$) est un couple $d = (B, C)$ où $B$ est un ensemble de blocs sur $E$ et $C$ est un ensemble de connecteurs sur $B$. Dans la suite, nous notons $\mathtt{BD}(E)$ l'ensemble des diagrammes de blocs sur $E$.

Nous avons annoncé que les diagrammes de blocs permettaient de représenter des programmes temps réel. Précisons d'abord que nous considérons des programmes temps réel ayant un temps discrétisé et infini. De tels programmes considèrent des flux d'entrées qui vont varier au cours du temps et produisent des flux de sorties. Formellement, un flux $x$ (défini sur $D$) est une suite infinie de valeurs dans $D$, et $x(t)$ donne la valeur au temps $t$ du flux $x$ ($t \in \mathbb{N}$). Nous notons $\mathbb{S}(D)$ l'ensemble des flux sur $D$. De fait, à partir de maintenant, nous considérons des diagrammes de blocs définis sur des flux, c'est-à-dire des diagrammes dont les opérateurs des blocs prennent des flux en entrée et retournent des flux. Ainsi nous définissons deux types de blocs : les blocs fonctionnels et les blocs temporels. Un bloc est dit fonctionnel ssi il existe une fonction qui, pour tous les pas de temps $t$, donne la valeur au temps $t$ des flux de sortie, en fonction uniquement des valeurs au temps $t$ des flux en entrée. C'est à dire que l'opérateur d'un bloc fonctionnel ne dépend que du temps présent et pas des temps passés. Au contraire, un bloc est dit temporel ssi il utilise des valeurs des flux à des temps passés pour calculer les sorties au temps présent. Nous appelons *exécution* d'un diagramme de blocs une affectation de flux aux entrées et aux sorties des blocs qui satisfait les opérateurs. L'exemple 1 présente un diagramme de blocs sur des flux.

**Exemple 1.** La figure 1 présente un diagramme de blocs $d$ sur des flux. $d$ contient 5 blocs fonctionnels : 0, 0.1, 0.9, + et ×. Les blocs 0, 0.1 et 0.9 sont constants (c'est-à-dire $\forall t \in \mathbb{N} : 0.9(t) = 0.9$). Le bloc + (resp. ×) est tel que pour deux flux à nombres réels $a$, $b$ il produit le flux de sortie $c$ vérifiant $c(t) = a(t) + b(t)$ (resp. $c(t) = a(t) \times b(t)$)

pour tout $t \in \mathbb{N}$. $d$ contient également un bloc temporel : le bloc *followed by* abrégé par *fby* Par convention, nous hachurons les blocs temporels. La présence du bloc temporel permet de casser la dépendance cyclique. Ainsi, ce diagramme de blocs admet une seule exécution. Les valeurs pour les quatre premiers temps sont inscrites entre crochets à côté des connecteurs (remarquez le délai dû au bloc *fby*).

Après avoir décrit les diagrammes de blocs et leurs exécutions, nous présentons la problématique de vérification sous-jacente. La présence de circuits dans la représentation graphique des diagrammes de blocs ainsi que les branchements de flux d'entrées faiblement bornés peuvent générer des exécutions plus ou moins variées. Un des souhaits récurrents de la vérification est de borner les valeurs prises par les variables d'un programme. Traduit dans le contexte des diagrammes de blocs, cela revient à chercher quelles valeurs peuvent passer par les entrées et les sorties des blocs qui composent un diagramme de blocs. Ainsi, le problème de vérification que nous considérons est celui de trouver pour chaque entrée et sortie de bloc un intervalle, appelé *sur-approximation*, qui contienne l'ensemble des valeurs prises sur cette entrée ou sortie pour toutes les exécutions possibles du diagramme de blocs. Nous appelons ce problème, le problème de sur-approximation de flux.

## Modélisation et résolution en contraintes

Nous proposons une modélisation et une résolution en contraintes du problème de sur-approximation de flux dans les diagrammes de blocs. La première remarque est que nous restreignons l'usage des blocs temporels au bloc *fby* uniquement. Ainsi, sous cette condition, tout cycle dans un diagramme de blocs admet au moins un bloc *fby*.[1]

Nous rappelons que modéliser en contraintes revient à transformer un problème à résoudre en un problème de satisfaction/d'optimisation de contraintes. Pour le problème qui nous intéresse, nous souhaitons trouver pour chaque entrée et sortie de bloc un intervalle qui sur-approxime l'ensemble des valeurs des flux pouvant passer par cette entrée/sortie. L'approche que nous avons choisie est de trouver une sur-approximation la plus petite possible sans imposer le critère de minimalité. Nous verrons par la suite que les résultats que nous obtenons sont très satisfaisants, proches du minimum. Pour rappel, un CSP est défini comme un ensemble de variables $X$ chacune associée à un domaine $D_x$ ($x \in X$) et un ensemble de contraintes $C$. Les contraintes définissent les valuations des variables acceptées par le CSP (ex : la contrainte $x = 2 + y$ porte sur deux variables $x$ et $y$ et impose l'égalité entre la valeur de $x$ et la valeur de $y$ plus 2).

Dans cette thèse, nous procédons en trois étapes de raffinement de nos modèles en contraintes pour atteindre notre modélisation finale. Le premier modèle, appelé modèle naïf, transforme le diagramme de blocs en un réseau de contraintes où les variables du

[1]Nous expliquons dans la thèse que cette restriction n'est pas trop forte et que les blocs temporels usuels (delay, memory, $n$-delay) peuvent être réécrits avec le bloc *fby*.

```
a, b, c, d, e, f ∈ 𝕊(ℝ)

a = fby(e, c)
b = ×(a, f)
c = +(b, d)
d = 0.1
e = 0
f = 0.9
```

Figure 2: CSP $\mathcal{C}_1$ produit par notre modèle en contraintes naïf pour le diagramme de blocs de la figure 1

```
a, b, c, d, e, f ∈ 𝕀(ℝ̄)

a = [fby](e, c)
b = [×](a, f)
c = [+](b, d)
d = [0.1]
e = [0]
f = [0.9]
```

Figure 3: CSP $\mathcal{C}_2$ produit par notre modèle en contraintes intermédiaire pour le diagramme de la figure 1

```
a, b, c, d, e, f ∈ 𝕀(ℝ̄)

d = 0.1
e = 0
f = 0.9

real-time-loop([a=[fby](e, c); b=[×](a, f); c=[+](b, d)],
               [d; e; f],
               [])
```

Figure 4: CSP $\mathcal{C}_3$ produit par notre modèle en contraintes final pour le diagramme de blocs de la figure 1

CSP correspondent aux entrées et aux sorties des blocs du diagramme de blocs, et les contraintes sont exactement les opérateurs des blocs. Ainsi, toute solution de ce CSP correspond à une exécution du diagramme de blocs (c'est-à-dire les domaines des variables sont l'ensemble des flux à valeurs dans $D$). Les flux solutions pouvant être infinis et le nombre de solutions pouvant également être infini, il y a peu d'espoir de parvenir à synthétiser par ce modèle en contraintes l'ensemble des flux solutions pour chaque variable via une sur-approximation dans les intervalles. De fait, en s'inspirant de l'interprétation abstraite, nous considérons une abstraction du problème (qui peut être vue comme une relaxation pour la communauté contrainte) pour construire un deuxième modèle en contraintes. Ce second modèle, appelé modèle intermédiaire, considère comme domaine des variables l'ensemble des intervalles fermés à bornes dans $\overline{D}$.[2] Les opérateurs des blocs sur les flux sont remplacés par une de leurs extensions aux intervalles. Ce modèle est tel que toute solution répond au problème de la sur-approximation de flux. Cependant, ce modèle intermédiaire retourne des solutions de faible qualité et il reste facilement bloqué aux infinis lors de la propagation de contraintes. Dès lors, nous proposons un dernier modèle, appelé modèle final, prenant en compte ce défaut de filtrage. Pour ce faire, nous introduisons une nouvelle contrainte globale : la contrainte `real-time-loop` . Nos recherches ont montré que la difficulté se posait au niveau des circuits. Chaque contrainte `real-time-loop` contient l'ensemble de la sémantique d'un circuit. C'est grâce à la connaissance de la sémantique haut niveau de cette contrainte que nous avons proposé un algorithme de filtrage dédié offrant des sur-approximations de meilleure qualité.

[2] $\overline{D}$ appelé ensemble étendu de $D$ correspond à l'union de $D$ et de ses limites (*e.g.*, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$)

L'algorithme de filtrage proposé est inspiré de l'interprétation abstraite et plus particulièrement de la technique de recherche d'invariant inductif par la méthode des montées et descentes dans le treillis des sur-approximations (*widening and narrowing technique).* L'algorithme proposé n'assure pas de retourner une sur-approximation minimale. Cependant il possède des heuristiques permettant d'augmenter les possibilités de recherche et donc d'augmenter les chances de s'approcher de la solution minimale. L'exemple 2 illustre les différences entre le modèle naïf, intermédiaire et final.

**Exemple 2.** La figure 2 contient le CSP $\mathcal{C}_1$ résultant de notre modélisation en contrainte naïve pour le diagramme de blocs $d$ présenté dans la figure 1. Notez en premier, que chaque connecteur a été associé à une variable. Par exemple, les sorties des blocs constants 0.1, 0 et 0.9 correspondent aux variables $d$, $e$ et $f$ dans le CSP. De plus, chaque bloc produit une contrainte dans le CSP (nous utilisons une notation préfixée pour l'écriture des opérateurs en contraintes). Par exemple le bloc *fby* du diagramme de blocs $d$ produit la contrainte $a = fby(e, c)$ dans $\mathcal{C}_1$. Ainsi, nous avons que toute solution de $\mathcal{C}_1$ correspond à une exécution de $d$. Ensuite, la figure 3 contient le CSP $\mathcal{C}_2$ produit par notre modélisation en contrainte intermédiaire pour le diagramme de blocs $d$. Étant donné un opérateur sur des flux, la notation $[f]$ correspond à l'extension aux intervalles en tant que contrainte de la fonction $f$. Par exemple, $a = [0, 1]$, $b = [2, 3]$ et $c = [2, 4]$ satisfait la contrainte $a[+]b = c$. De fait, le passage de $\mathcal{C}_1$ à $\mathcal{C}_2$ a remplacé les domaines de flux à des domaines à intervalles, et les opérateurs dans les contraintes sont remplacés par leurs extensions aux intervalles. Ainsi, pour toute variable $x$ de $\mathcal{C}_1$, pour toute solution de $\mathcal{C}_2$ l'intervalle choisi pour $x$ contient toutes les valeurs des flux solutions de $\mathcal{C}_1$ pour la variable $x$ (c'est-à-dire pour toute valuation $v_2$ solution de $\mathcal{C}_2$ et pour toute valuation $v_1$ solution de $\mathcal{C}_1$ nous avons $v_1(x)(t) \in v_2(x)$ pour tout $t \in \mathbb{N}$). Enfin, la figure 4 contient le CSP $\mathcal{C}_3$ utilisant notre contrainte globale `real-time-loop` . Cette contrainte prend trois arguments qui sont, dans l'ordre : les contraintes formant le circuit, les variables d'entrées du circuit, les variables de sorties du circuit. Ainsi dans $\mathcal{C}_1$ la contrainte `real-time-loop` contient les blocs *fby*, $\times$ et $+$, les entrées $d$, $e$ et $f$ et n'a pas de sorties.

Pour évaluer notre modélisation en contraintes, nous avons choisi le langage FAUST permettant de faire la synthèse et le traitement temps réel de flux audio. Ce langage possède une sémantique bien définie de telle sorte que le compilateur FAUST peut générer pour chaque programme le diagramme de blocs qui en est la sémantique. Ainsi, nous avons considéré un ensemble de programmes de la bibliothèque FAUST et nous les avons modélisés en contraintes en utilisant notre encodage final. Nous avons ensuite utilisé le solveur de contraintes continues IBEX pour y implanter notre contrainte globale `real-time-loop` . L'objectif était de vérifier que les flux de sorties des programmes ne provoquaient pas de saturation (c'est-à-dire qu'ils étaient sur-approximés par l'intervalle $[-1; 1]$). Nos résultats montrent que nous parvenons dans la majeure partie des cas à trouver la plus petite sur-approximation dans les intervalles et dans des temps très courts

(de l'ordre de la seconde). Ainsi notre approche a été prise en considération par les développeurs de FAUST pour une éventuelle intégration dans une version future du logiciel. Pour autant, il est important de noter que la plus petite sur-approximation dans les intervalles par le calcul intervalle est parfois très éloignée du plus petit intervalle contenant les valeurs prises par les flux. En effet, les calculs à partir de sur-approximations produisent des marges de sur-approximations qui se répètent et s'amplifient. Il existe également des opérateurs de blocs produisant de grandes imprécisions lorsqu'ils sont étendus aux intervalles (ex : $X[-]X \neq \{0\}$).

Pour conclure, nous avons proposé plusieurs modélisations en contraintes pour le problème de la sur-approximation de flux dans les diagrammes de blocs. Nous avons proposé la contrainte globale `real-time-loop` pour répondre au problème donné. Puis nous avons évalué avec succès notre approche en considérant le problème de recherche de saturation dans des programmes FAUST. Ces travaux ont montré l'intérêt de l'utilisation de techniques de programmation par contraintes dans des cadres exotiques (la vérification de programmes utilisant des variables de flux). Ces travaux ont fait l'objet de trois communications/publications [1, 2, 3].

## Vérification en contraintes de systèmes probabilistes

Les chaînes de Markov (MCs) sont largement utilisées pour modéliser une très grande variété de systèmes basés sur des transitions probabilistes (ex : protocoles aléatoires, systèmes biologiques, environnements financiers). D'un autre côté, les chaînes de Markov à intervalles paramétrés (pIMCs) sont un formalisme de spécification permettant de représenter de façon compacte des ensembles infinis de chaînes de Markov. En effet, les PIMCs prennent en compte l'imprécision ou le manque de connaissances quant à la probabilité exacte de chaque évènement/transition du système en considérant des intervalles paramétrés de probabilités. Dans la seconde contribution de cette thèse (Chapitre 5), nous proposons d'abord une comparaison formelle de trois sémantiques existantes pour les PIMCs. Ensuite, nous proposons des encodages en contraintes pour vérifier des propriétés d'accessibilité qualitative et quantitative sur les pIMCs. En particulier, l'étude formelle des différentes sémantiques des pIMCs a permis de proposer des encodages en contraintes succincts et performants. Enfin, nous concluons avec des expériences montrant l'amélioration de nos encodages en contraintes par rapport à ceux de l'état de l'art résolvant les mêmes problèmes sur les pIMCs.

### Contexte et problématique

Un processus aléatoire est un système dans lequel le passage d'un état à un autre état est probabiliste : chaque état successeur a une certaine probabilité d'être choisi. Une chaîne de Markov à temps discret est un processus aléatoire dont le passage d'un état à un autre

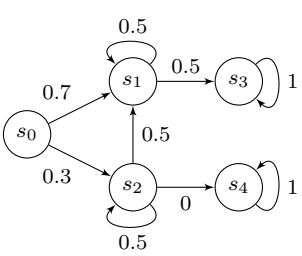



Figure 5: Exemple de MC

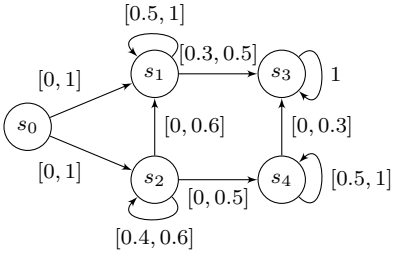



Figure 6: Exemple d'IMC

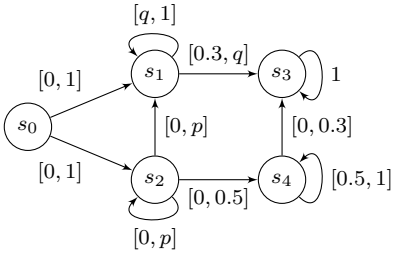



Figure 7: Exemple d'IMC

se fait par pas de temps. Ce changement d'états est décrit par une fonction de transition qui à chaque état associe une distribution de probabilités sur ses états successeurs. Une chaîne de Markov peut être vue comme un graphe dirigé dont les noeuds correspondent aux états de la MC et dont les arêtes sont étiquetées par les probabilités de la fonction de transition. Avec cette représentation, une transition manquante entre deux états vaut pour une probabilité de 0. Dans cette thèse nous considérons des chaînes de Markov ayant un ensemble d'états fini.

Modéliser une application comme une chaîne de Markov suppose de connaître exactement les probabilités pour chaque transition du système. Cependant, ces quantités peuvent être difficiles à calculer ou à mesurer pour des applications réelles (ex : erreurs de mesure, connaissance partielle). Les chaînes de Markov à intervalles (IMCs) étendent les chaînes de Markov en autorisant les probabilités de transition à varier dans des intervalles donnés. Ainsi, à chaque transition d'état à état est associé un intervalle au lieu d'une probabilité exacte.

Enfin, les chaînes de Markov à intervalles paramétrés (pIMCs) autorisent l'utilisation d'intervalles dont les bornes sont variables. Ces bornes variables sont alors représentées par des paramètres (ou des combinaisons de paramètres), ce qui permet notamment l'expression de dépendances entre plusieurs transitions du système. Ainsi, les pIMCs représentent, d'une manière compacte et avec une structure finie, un ensemble potentiellement infini d'IMCs. Par transitivité, les pIMCs permettent de représenter potentiellement une infinité d'ensembles de chaînes de Markov.

La propriété que nous allons vérifier est celle de l'accessibilité (en : *reachability*) dans les MCs. Formellement, la probabilité d'atteindre un état dans une MCs est donnée par la somme de la probabilité de tous les chemins atteignant l'état désiré (c'est-à-dire tous les chemins finis partant de l'état initial, terminant par l'état désiré et ne rencontrant pas cet état avant). De plus, la probabilité d'un chemin correspond aux produits des probabilités rencontrées sur les transitions état à état. Nous notons $\mathbb{P}^{\mathcal{M}}(\Diamond s)$ la probabilité d'atteindre un état $s$ dans une MC $\mathcal{M}$.

**Exemple 3.** La figure 5 représente une MC $\mathcal{M}$ avec 5 états $s_0$, $s_1$, $s_2$, $s_3$ et $s_4$ où $s_0$ est l'état initial et où nous pouvons lire par exemple que la probabilité de passer de l'état $s_0$

à l'état $s_1$ vaut 0.7 et que celle de passer de l'état $s_0$ à $s_2$ vaut 0.3. Ainsi les séquences d'états $(s_0, s_1, s_3)$, $(s_0, s_3)$ et $(s_0, s_2, s_1, s_3)$ sont trois chemins (finis) partant de l'état initial $s_0$ et terminant dans l'état $s_3$ ayant pour probabilités respectives $0.7 \cdot 0.5 = 0.35$, $0.7 \cdot 0 = 0$ et $0.3 \cdot 0.5 \cdot 0.5 = 0.075$ . Enfin, la probabilité d'atteindre l'état $s_1$ vaut $p(s_0)(s_1) + \Sigma_{i=0}^{+\infty} p(s_0)(s_2) \cdot p(s_2)(s_2)^i \cdot p(s_2)(s_1) = p(s_0)(s_1) + p(s_0)(s_2) \cdot p(s_2)(s_1) \cdot (1/(1 - p(s_2)(s_2))) = 1$. A côté, la figure 6 représente une IMC $\mathcal{I}$. Puisque $\mathcal{M}$ a la même structure que $\mathcal{I}$ et que les probabilités des transitions de $\mathcal{M}$ appartiennent aux intervalles correspondants dans $\mathcal{I}$ nous disons que $\mathcal{M}$ satisfait $\mathcal{I}$. Pour terminer, la figure 7 représente une pIMC utilisant deux paramètres $p$ et $q$. Notons que choisir les valeurs 0.6 pour $p$ et 0.5 pour $q$ produit l'IMC $\mathcal{I}$. Nous disons que $\mathcal{I}$ implémente $\mathcal{P}$. De fait, puisque $\mathcal{M}$ satisfait $\mathcal{I}$ et que $\mathcal{I}$ implémente $\mathcal{P}$ nous disons que $\mathcal{M}$ satisfait $\mathcal{P}$.

Les pIMCs et les IMCs sont appelées des modèles d'abstractions de chaînes de Markov. En effet, comme dit précédemment, tout pIMC ou IMC représente/abstrait un ensemble de chaînes de Markov. Ainsi, nous disons qu'une chaîne de Markov satisfait une abstraction de chaînes de Markov ssi la chaîne de Markov appartient à l'ensemble des MCs représentées par l'abstraction. De plus, les IMCs sont formellement définies avec trois sémantiques d'abstractions : 1) *once-and-for-all*, 2) *IMDP* et 3) *at-every-step*. La première sémantique définit que l'ensemble des MCs qui satisfont une IMC sont celles qui ont la même structure que l'IMC et dont la probabilité $p$ de passer d'un état $s$ accessible à un état $s'$ appartient à l'intervalle de probabilités sur la transition de $s$ vers $s'$ dans l'IMC. Nous disons que pour chaque intervalle de l'IMC une et une seule probabilité est sélectionnée. La seconde sémantique définit que l'ensemble des MCs qui satisfont une IMC sont celles qui autorisent de choisir plusieurs probabilités pour un même intervalle d'une IMC. Nous disons que l'IMC originale est "dépliée". Ainsi, un état d'une IMC peut se retrouver "copié" plusieurs fois dans la MC qui satisfait l'IMC. Enfin, la troisième sémantique autorise sous certaines conditions que certains états de l'IMC peuvent être fusionnés ou scindés en plusieurs états tout en autorisant le dépliage de l'IMC. Cette sémantique correspond à la sémantique originelle donnée aux IMCs. Nous montrons dans cette thèse que la sémantique *at-every-step* est plus générale que la *IMDP*, qui est plus générale que la *once-and-for-all*. Toutes ces sémantiques s'étendent aux pIMCs.

Ainsi, la partie contribution aborde trois problèmes majeurs de vérification sur les pIMCs : la *consistance (existentielle)*, l'*accessibilité qualitative (existentielle)* et l'*accessibilité quantitative (existentielle)*. Le problème de la consistance d'une pIMC détermine si une pIMC admet au moins une MC qui la satisfait. Le problème de l'accessibilité qualitative détermine si pour un ensemble d'états à atteindre il existe une MC qui satisfait la pIMC où un des états but peut être atteint (c'est-à-dire qu'il existe un chemin avec une probabilité non nulle qui part de l'état initial de cette MC et atteint l'état but). Le problème de l'accessibilité quantitative détermine si, pour un ensemble d'états à atteindre et un seuil d'accessibilité, il existe une MC qui satisfait la pIMC où la

probabilité d'atteindre les états buts est supérieure ou inférieure au seuil.

## Modélisation et résolution en contraintes

Dans un premier temps, nous avons prouvé que les valeurs de probabilités maximales et minimales d'accessibilité d'états sont atteintes par les MCs de même structure que les IMCs/pIMCs. C'est grâce à ce théorème fort que nous avons pu proposer des modèles en contraintes succincts pour vérifier les problèmes présentés dans la précédente section. En effet, il n'est plus nécessaire de considérer toutes les MCs avec "dépliages" pour vérifier la consistance et les propriétés d'accessibilité qualitatives et quantitatives, mais uniquement les MCs de même structure que la pIMC. Nous présentons maintenant les modèles en contraintes proposés. Nos modélisations en contraintes sont modulaires. C'est-à-dire qu'un premier lot de contraintes résout le problème de la consistance, puis l'ajout d'un second lot de contraintes vient répondre au problème de l'accessibilité qualitative et l'ajout d'un dernier lot permet de répondre au problème de l'accessibilité quantitative. L'objectif de nos modèles en contraintes est de construire une chaîne de Markov qui satisfasse l'IMC vérifiant la propriété désirée (consistance, accessibilité qualitative ou accessibilité quantitative). Ainsi, nos modèles en contraintes encodent de telles MCs. Formellement, étant donnée une pIMC à vérifier et $T$ un ensemble d'états à atteindre, nos modèles en contraintes définissent les variables $\rho_s$, $\omega_s$, $\alpha_s$, $\lambda_s$, $\pi_s$ pour chaque état $s$ de la pIMC, une variable $\phi_p$ par paramètre $p$ de la pIMC et une variable $\theta_s^{s'}$ par intervalle paramétré dans la pIMC. Rappelons que ces variables ont pour objectif de construire une MC. Chaque variable $\theta_s^{s'}$ détermine la probabilité de la transition allant de l'état $s$ vers l'état $s'$ dans la MC. Pour tout état $s$, la variable $\rho_s$ est une variable booléenne indiquant si l'état $s$ est accessible depuis l'état initial ; la variable $\omega_s$ est une variable entière qui vaut $k$ s'il existe un chemin de taille $k-1$ depuis l'état initial vers $s$, et qui vaut 0 sinon ; la variable $\alpha_s$ est une variable entière qui vaut $k$ s'il existe un chemin de taille $k-1$ depuis $s$ vers un état but $s'$ dans $T$, et qui vaut 0 sinon ; la variable $\lambda_s$ est une variable booléenne qui vaut $\mathsf{true}$ ssi il existe un chemin depuis l'état initial vers un état but de $T$ passant par $s$ ; et la variable $\pi_s$ vaut la probabilité d'atteindre l'état $s$ depuis l'état initial si $s$ est accessible et qui vaut 0 sinon. Voici les contraintes à considérer pour chaque état $s$ de la pIMC :

**(1)** $\rho_s$, si $s = s_0$

**(2)** $\neg\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Pred}(s)\setminus\{s\}} \theta_{s'}^{s} = 0$, si $s \neq s_0$

**(3)** $\neg\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Succ}(s)} \theta_s^{s'} = 0$

**(4)** $\rho_s \Leftrightarrow \Sigma_{s' \in \mathtt{Succ}(s)} \theta_s^{s'} = 1$

**(5)** $\rho_s \Rightarrow \theta_s^{s'} \in P(s, s')$, pour tout $s' \in \mathtt{Succ}(s)$

---

**(6)** $\omega_s = 1$, si $s = s_0$

**(7)** $\omega_s \neq 1$, si $s \neq s_0$

**(8)** $\rho_s \Leftrightarrow (\omega_s \neq 0)$

**(9)** $\omega_s > 1 \Rightarrow \bigvee_{s' \in \texttt{Pred}(s) \setminus \{s\}} (\omega_s = \omega_{s'} + 1) \wedge (\theta_s^{s'} > 0)$, si $s \neq s_0$

**(10)** $\omega_s = 0 \Leftrightarrow \bigwedge_{s' \in \texttt{Pred}(s) \setminus \{s\}} (\omega_{s'} = 0) \vee (\theta_s^{s'} = 0)$, si $s \neq s_0$

---

**(11)** $\alpha_s = 1$, si $s \in T$

**(12)** $\alpha_s \neq 1$, si $s \notin T$

**(13)** $\lambda_s \Leftrightarrow (\rho_s \wedge (\alpha_s \neq 0))$

**(14)** $\alpha_s > 1 \Rightarrow \bigvee_{s' \in \texttt{Succ}(s) \setminus \{s\}} (\alpha_s = \alpha_{s'} + 1) \wedge (\theta_s^{s'} > 0)$, si $s \notin T$

**(15)** $\alpha_s = 0 \Leftrightarrow \bigwedge_{s' \in \texttt{Succ}(s) \setminus \{s\}} (\alpha_{s'} = 0) \vee (\theta_s^{s'} = 0)$, si $s \notin T$

---

**(16)** $\neg\lambda_s \Rightarrow \pi_s = 0$

**(17)** $\lambda_s \Rightarrow \pi_s = 1$, si $s \in T$

**(18)** $\lambda_s \Rightarrow \pi_s = \Sigma_{s' \in \texttt{Succ}(s)} \pi_{s'} \theta_{s'}^s$, si $s \notin T$

Nous montrons dans cette thèse que ces lots de contraintes permettent de répondre aux problèmes de la consistance existentielle (contraintes **(1)** à **(5)**), l'accessibilité qualitative existentielle (contraintes **(1)** à **(10)**) et l'accessibilité quantitative existentielle (contraintes **(1)** à **(18)**). De plus, nos modèles sont linéaires en taille par rapport à la taille de la pIMC là où les modèles de l'état de l'art sont exponentiels en taille. Nous terminons la contribution par une évaluation pratique de nos modèles en contraintes. Les contraintes sont linéaires (sauf la contrainte **(18)** qui est quadratique) et utilisent des expressions logiques comme l'équivalence et l'implication. Quant aux variables, nous sommes dans le cas mixte avec la présence de variables booléennes, entières et réelles. Ainsi, la communauté Satisfiability Modulo Theory se propose de résoudre ce genre de problèmes. Dans le cas linéaire il y a également la communauté Mixed Integer Linear Programming qui accepte nos CSPs. Nous sommes allés chercher un jeu de tests dans la communauté des MCs. Nous avons étendu ces MCs à des pIMCs et avons vérifié dessus les propriété de consistance, d'accessibilités qualitative et quantitative. Notre outil est disponible en ligne.[3] Nos résultats montrent que nos modèles en contraintes sont plus performants que ceux de l'état de l'art. En effet, nos modèles en contraintes gagnent un ordre de complexité en terme de taille ce qui permet de s'attaquer à des pIMCs beaucoup plus grandes (c'est-à-dire avec des dizaines de milliers d'états). Enfin, nous proposons un

[3] https://github.com/anicet-bart/pimc_pylib

premier outil pour réaliser la vérification d'accessibilité quantitative sur des pIMCs. Pour cette propriété, nous parvenons à traiter des pIMCs ayant une centaine d'états.

Pour conclure, dans cette partie de la thèse, nous avons réalisé une analyse formelle de propriétés sur les abstractions de chaînes de Markov. Dans cette analyse, nous avons montré que les différentes sémantiques données aux IMCs sont équivalentes par rapport à l'accessibilité quantitative de probabilité maximale et minimale (ce qui s'étend aux pIMCs). Grâce à ce résultat, nous avons présenté des modèles en contraintes qui forment la première solution pratique au problème de l'accessibilité quantitative dans les pIMCs. Dans le même temps, nous avons amélioré les encodages en contraintes existants pour résoudre la consistance existentielle et l'accessibilité qualitative. Enfin, nous avons proposé un outil implémentant nos divers encodages en contraintes. Ces travaux ont fait l'objet de trois communications/publications [4, 5, 6].

# Conclusion

Dans cette thèse, nous avons abordé deux familles de problèmes traitant de la vérification de programmes. Pour chaque cas, nous avons d'abord étudié formellement la nature des problèmes de vérification concernés avant de proposer une résolution par les contraintes. Puisque nous ne nous imposions aucune restriction concernant le langage de contraintes, nous avons proposé des modélisations par contraintes utilisant des contraintes non-linéaires sur des variables non bornées, des variables mixtes entières/linéaires sur des contraintes linéaires, mais aussi des contraintes quadratiques sur des variables mixtes. Ainsi, la vérification de programmes est un champ de recherche riche pouvant faire appel à divers outils de la programmation par contraintes. La complexité théorique du problème de la vérification de programmes, comme celle du problème de la satisfaction de contraintes, peut s'avérer élevée. Cependant, les solveurs de la programmation par contraintes peuvent résoudre en partie ces problèmes difficiles. Pour autant, les communautés de la programmation par contraintes avancent sur des axes de recherches séparés en développant des solveurs dédiés à des langages de contraintes spécifiques. Finalement, dans cette thèse nous avons abordé la vérification de programmes sous l'angle de la programmation par contraintes. Cela nous a permis d'apporter de nouvelles idées dans les processus de vérification de programmes et de rapprocher ces deux domaines de recherche.

# Bibliography

[1] Anicet Bart, Charlotte Truchet, and Eric Monfroy. “Contraintes sur des flux appliquées à la vérification de programmes audio”. In: *Onzièmes Journées Francophones de Programmation par Contraintes (JFPC 2015)*. 2015.

[2] Anicet Bart, Charlotte Truchet, and Eric Monfroy. “Verifying a Real-Time Language with Constraints”. In: *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*. 2015, pp. 844–851.

[3] Anicet Bart, Charlotte Truchet, and Eric Monfroy. “A global constraint for over-approximation of real-time streams”. In: *Constraints* 22.3 (2017), pp. 463–490.

[4] Anicet Bart et al. “Vérification de chaînes de Markov à intervalles paramétrés avec des contraintes”. In: *Treizèmes Journées Francophones de Programmation par Contraintes (JFPC 2017)*. 2017.

[5] Anicet Bart et al. “An Improved Constraint Programming Model for Parametric Interval Markov Chain Verification”. Third Workshop ”CP meets Verification 2016” (no proceedings) at the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016). 2016.

[6] Anicet Bart et al. “Reachability in Parametric Interval Markov Chains using Constraints”. In: *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*. 2017.

[7] Alexander Bockmayr and Volker Weispfenning. “Solving Numerical Constraints”. In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 751–842.

[8] Peter Gács and Laszlo Lovász. “Khachiyan’s algorithm for linear programming”. In: *Mathematical Programming at Oberwolfach*. Ed. by H. König, B. Korte, and K. Ritter. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 61–68.

[9] Thomas J. Schaefer. "The Complexity of Satisfiability Problems". In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: ACM, 1978, pp. 216–226.

[10] R. Fourer, D. M. Gay, and B. Kernighan. "Algorithms and Model Formulations in Mathematical Programming". In: ed. by Stein W. Wallace. New York, NY, USA: Springer-Verlag New York, Inc., 1989. Chap. AMPL: A Mathematical Programming Language, pp. 150–151.

[11] *IBM ILOG CPLEX Optimizer*. urlhttp://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Last 2010.

[12] Pietro Belotti et al. "Mixed-integer nonlinear optimization". In: *Acta Numerica* 22 (2013), pp. 1–131.

[13] Gecode Team. *Gecode: Generic Constraint Development Environment*. Available from `http://www.gecode.org`. 2017.

[14] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP - Java Constraint Programming Solver*. 2013.

[15] *Satisfiability Suggested Format*. Tech. rep. 1993.

[16] Steven David Prestwich. "CNF Encodings". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 75–97.

[17] Rutherford Aris. *Mathematical modelling techniques*. Dover books on computer science. New York, NY: Dover, 1995.

[18] Nicolas Beldiceanu et al. "Global Constraint Catalog: Past, Present, and Future". In: *Constraints* 12.1 (2007), pp. 21–62.

[19] Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. "XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems". In: *CoRR* (2016).

[20] Ralph Becket. *Specification of FlatZinc*. URL: `http://www.minizinc.org/downloads/doc-1.3/flatzinc-spec.pdf` (visited on 05/08/2017).

[21] Christophe Lecoutre. "MCSP3 : la modélisation pour tous". In: *Treizèmes Journées Francophones de Programmation par Contraintes (JFPC 2017)*. 2017.

[22] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming – CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[23] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006.

[24] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[25] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782.

[26] Harry R. Lewis. "Satisfiability Problems for Propositional Calculi". In: *Mathematical Systems Theory* 13 (1979), pp. 45–53.

[27] M. K. Kozlov, S. P. Tarasov, and L. G. Khachiyan. "Polynomial Solvability of Convex Quadratic Programming". In: *Doklady Akademiia Nauk SSSR* 248 (1979).

[28] A. H. Land and A. G. Doig. "An automatic method for solving discrete programming problems". In: *ECONOMETRICA* 28.3 (1960), pp. 497–520.

[29] Jens Clausen. *Branch and Bound Algorithms – Principles And Examples*. 1999.

[30] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782.

[31] Frédéric Benhamou et al. "Revising Hull and Box Consistency". In: *International Conference on Logic Programming*. MIT press, 1999, pp. 230–244.

[32] David Monniaux. "A Survey of Satisfiability Modulo Theory". In: *Computer Algebra in Scientific Computing: 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*. Ed. by Vladimir P. Gerdt et al. Cham: Springer International Publishing, 2016, pp. 401–425.

[33] David R. Morrison et al. "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discrete Optimization* 19 (2016), pp. 79–102.

[34] Peter van Beek. "Backtracking Search Algorithms". In: *Handbook of Constraint Programming*. 2006, pp. 85–134.

[35] Jean-Guillaume Fages, Gilles Chabert, and Charles Prud'homme. "Combining finite and continuous solvers". In: *CoRR* abs/1402.1361 (2014).

[36] Tallys H. Yunes, Ionut D. Aron, and John N. Hooker. "An Integrated Solver for Optimization Problems". In: *Operations Research* 58.2 (2010), pp. 342–356.

[37] Milan Bankovic and Filip Maric. "An Alldifferent Constraint Solver in SMT". In: *Proceedings of the 2010 SMT Workshop*.

[38] Farhad Arbab and Eric Monfroy. "Coordination of Heterogeneous Distributed Cooperative Constraint Solving". In: *SIGAPP Appl. Comput. Rev.* 6.2 (Sept. 1998), pp. 4–17. ISSN: 1559-6915.

[39] Eric Monfroy, Michaël Rusinowitch, and René Schott. "Implementing Non-linear Constraints with Cooperative Solvers". In: *Proceedings of the 1996 ACM Symposium on Applied Computing*. SAC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 63–72. ISBN: 0-89791-820-7.

[40] Laurent Granvilliers, Eric Monfroy, and Frédéric Benhamou. "Symbolic-interval Cooperation in Constraint Programming". In: *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*. ISSAC '01. London, Ontario, Canada: ACM, 2001, pp. 150–166. ISBN: 1-58113-417-7.

[41] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. "Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 228–245.

[42] Frédéric Lardeux and Eric Monfroy. "Expressively Modeling the Social Golfer Problem in SAT". In: *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavik, Iceland, 1-3 June, 2015, 2014*. 2015, pp. 336–345.

[43] David Cohen et al. "Symmetry Definitions for Constraint Satisfaction Problems". In: *Principles and Practice of Constraint Programming - CP 2005: 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings*. Ed. by Peter van Beek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 17–31.

[44] Eugene Asarin et al. "Using Redundant Constraints for Refinement". In: *Automated Technology for Verification and Analysis: 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*. Ed. by Ahmed Bouajjani and Wei-Ngan Chin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 37–51.

[45] Paulraj S. and Sumathi P. "A comparative study of redundant constraints identification methods in linear programming problems." In: *Mathematical Problems in Engineering* 2010 (2010), Article ID 723402, 16 p.

[46] Jan Telgen. "On relaxation methods for systems of linear inequalities". In: *European Journal of Operational Research* 9.2 (1982), pp. 184–189.

[47] Frédéric Lardeux et al. "Set constraint model and automated encoding into SAT: application to the social golfer problem". In: *Annals OR* 235.1 (2015), pp. 423–452.

[48] "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985).

[49] Laurent Granvilliers and Frédéric Benhamou. "Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques". In: *ACM Trans. Math. Softw.* 32.1 (2006), pp. 138–156.

[50] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.

[51] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265.

[52] Amir Pnueli. "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

[53] P. Cousot and R. Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[54] Patrick Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[55] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN: 0471469122.

[56] Richard A. DeMillo and A. Jefferson Offutt. "Constraint-Based Automatic Test Data Generation". In: *IEEE Trans. Softw. Eng.* 17.9 (Sept. 1991), pp. 900–910. ISSN: 0098-5589.

[57] Arnaud Gotlieb. "Constraint-Based Testing: An Emerging Trend in Software Testing". In: *Advances in Computers*. Ed. by Atif Memon. ADCOM, UK: Academic Press. Vol. 99. Elsevier, Oct. 2015. Chap. 2, pp. 67–101.

[58] Krzysztof R. Apt. "From Chaotic Iteration to Constraint Propagation". In: *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*. 1997, pp. 36–55.

[59] Krzysztof R. Apt. "The Essence of Constraint Propagation". In: *Theor. Comput. Sci.* 221.1-2 (1999), pp. 179–210.

[60] Marie Pelleau et al. "A Constraint Solver Based on Abstract Domains". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 2013, pp. 434–454.

[61] Armin Biere et al. *Bounded Model Checking*. 2003.

[62] David Gerault, Marine Minier, and Christine Solnon. "Constraint Programming Models for Chosen Key Differential Cryptanalysis". In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*. 2016, pp. 584–601.

[63] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. "Program Verification as Satisfiability Modulo Theories". In: *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*. 2012, pp. 3–11.

[64] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking". In: *J. ACM* 52.3 (May 2005), pp. 365–473. ISSN: 0004-5411.

[65] Ugo Montanari. "Networks of Constraints: Fundamental Properties and Applications to Picture Processing". In: *Information Science* 7.2 (1974), pp. 95–132.

[66] Frédéric Benhamou and William J. Older. "Applying interval arithmetic to real, integer and Boolean constraints". In: *Journal of Logic Programming* 32.1 (1997), pp. 1–24.

[67] Gilles Chabert and Luc Jaulin. "Contractor programming". In: *Artificial Intelligence* 173.11 (2009), pp. 1079–1100.

[68] Y. Orlarey, D. Fober, and S. Letz. "An Algebra for Block Diagram Languages". In: *International Computer Music Conference*. 2002.

[69] Y. Orlarey, D. Fober, and S. Letz. "Syntactical and semantical aspects of Faust". English. In: *Soft Computing* 8.9 (2004), pp. 623–632. ISSN: 1432-7643. DOI: 10.1007/s00500-004-0388-1.

[70] N. Halbwachs et al. "The synchronous dataflow programming language Lustre". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[71] moForte. *audio modeling for mobile*. 2013. URL: http://www.moforte.com.

[72] Ignacio Araya et al. "Upper Bounding in Inner Regions for Global Optimization under Inequality Constraints". In: *Journal of Global Optimization* 2 (Oct. 2014), pp. 145–164.

[73] Andreas Podelski. "Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. Proceedings". In: ed. by Jens Palsberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. Chap. Model Checking as Constraint Solving, pp. 22–37. ISBN: 978-3-540-45099-3. DOI: 10.1007/978-3-540-45099-3_2.

[74] Byeongcheol Lee et al. "Compiler Construction: 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007. Proceedings". In: ed. by Shriram Krishnamurthi and Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. Chap. Correcting the Dynamic Call Graph Using Control-Flow Constraints, pp. 80–95.

[75] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. "An Efficient Algorithm for Parametric WCET Calculation". In: *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*. 2009, pp. 13–21.

[76] Hélène Collavizza et al. "Generating Test Cases Inside Suspicious Intervals for Floating-point Number Programs". In: *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. CSTVA 2014. Hyderabad, India: ACM, 2014, pp. 7–11. ISBN: 978-1-4503-2847-0.

[77] Olivier Ponsini, Claude Michel, and Michel Rueher. "Verifying floating-point programs with constraint programming and abstract interpretation techniques". In: *Automated Software Engineering* 23.2 (2016), pp. 191–217.

[78] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". In: *Journal of Logic and Computation* 2.4 (1992), pp. 511–547.

[79] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. "An abstract interpretation based combinator for modeling while loops in constraint programming". In: *Principles and Practice of Constraint Programming CP 2007, Providence, RI USA, September 23-27, 2007. Proceedings*. 2007, pp. 241–255.

[80] Olivier Ponsini, Claude Michel, and Michel Rueher. *Refining Abstract Interpretation-based Approximations with Constraint Solvers*. Tech. rep. Sept. 2011.

[81] Stefano Di Alesio et al. "Worst-Case Scheduling of Software Tasks - A Constraint Optimization Model to Support Performance Testing". In: *Principles and Practice of Constraint Programming CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. 2014.

[82] Benjamin Blanc et al. "Handling State-Machines Specifications with GATeL". In: *Electr. Notes Theor. Comput. Sci.* 264.3 (2010), pp. 3–17.

[83] JasperC.H. Lee and JimmyH.M. Lee. "Towards Practical Infinite Stream Constraint Programming: Applications and Implementation". English. In: *Principles and Practice of Constraint Programming*. Ed. by Barry OSullivan. Springer International Publishing, 2014. ISBN: 978-3-319-10427-0.

[84] Arnaud Lallouet et al. "Constraint programming on infinite data streams". In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*. AAAI Press. 2011.

[85] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-814757-4.

[86] Frédéric Benhamou and Laurent Granvilliers. "Continuous and Interval Constraints". In: *Handbook of Constraint Programming*. 2006, pp. 571–603. DOI: 10.1016/S1574-6526(06)80020-9. URL: http://dx.doi.org/10.1016/S1574-6526(06)80020-9.

[87] Olivier Lhomme. "Consistency techniques for numeric csps". In: 1993, pp. 232–238.

[88] Héléne Collavizza, FranÇois Delobel, and Michel Rueher. "A Note on Partial Consistencies over Continuous Domains". In: *Principles and Practice of Constraint Programming — CP98: 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings*. Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–161.

[89] Ramon Edgar Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs N. J., 1966.

[90] Julius O. Smith. *Spectral Audio Signal Processing*. online book, 2011 edition. URL: http://ccrma.stanford.edu/~jos/sasp/.

[91] Peter Schrammel. "Logico-Numerical Verification Methods for Discrete and Hybrid Systems". PhD thesis. University of Grenoble, 2013.

[92] James A Whittaker and Michael G Thomason. "A Markov chain model for statistical software testing". In: *IEEE Transactions on Software engineering* 20.10 (1994), pp. 812–824.

[93] Dirk Husmeier, Richard Dybowski, and Stephen Roberts. *Probabilistic Modeling in Bioinformatics and Medical Informatics*. Springer Publishing Company, Incorporated, 2010.

[94] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. "Parametric Real-time Reasoning". In: *STOC*. ACM, 1993, pp. 592–601.

[95] Bengt Jonsson and Kim Guldstrand Larsen. "Specification and Refinement of Probabilistic Processes". In: *LICS*. 1991, pp. 266–277.

[96] Krishnendu Chatterjee, Koushik Sen, and Thomas A. Henzinger. "Model-Checking omega-Regular Properties of Interval Markov Chains". In: *FOSSACS*. 2008, pp. 302–317.

[97] Koushik Sen, Mahesh Viswanathan, and Gul Agha. "Model-Checking Markov Chains in the Presence of Uncertainties". In: *Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006. Proceedings*. Ed. by Holger Hermanns and Jens Palsberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 394–410.

[98] Christian Dehnert et al. "PROPhESY: A PRObabilistic ParamEter SYnthesis Tool". In: *CAV*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 214–231.

[99] Ernst Moritz Hahn et al. "PARAM: A Model Checker for Parametric Markov Models". In: *CAV*. Vol. 6174. LNCS. Springer, 2010.

[100] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *CAV*. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[101] Marco Gribaudo, Daniele Manini, and Anne Remke, eds. *Model Checking of Open Interval Markov Chains*. Cham: Springer, 2015, pp. 30–42.

[102] Michael Benedikt, Rastislav Lenhardt, and James Worrell. "LTL model checking of interval Markov chains". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 32–46.

[103] Benoît Delahaye, Didier Lime, and Laure Petrucci. "Parameter Synthesis for Parametric Interval Markov Chains". In: *VMCAI*. 2016, pp. 372–390.

[104] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006. ISBN: 0444527265.

[105] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[106] Georg Cantor. "Uber unendliche, lineare Punktmannigfaltigkeiten V (On infinite, linear point-manifolds)". In: *Mathematische Annalen*. Vol. 21. 1883, pp. 545–591.

[107] Tichakorn Wongpiromsarn et al. "TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning". In: (2011).

[108] Alberto Puggelli et al. "Polynomial-Time Verification of PCTL Properties of MDPs with Convex Uncertainties". In: *CAV*. Springer, 2013, pp. 527–542.

[109] Benoît Delahaye. "Consistency for Parametric Interval Markov Chains". In: *SynCoP*. 2015, pp. 17–32.

[110] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *CAV*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[111] M. Kwiatkowska, G. Norman, and D. Parker. "Probabilistic Verification of Herman's Self-Stabilisation Algorithm". In: *Formal Aspects of Computing* 24.4 (2012), pp. 661–670.

[112] G. Norman and V. Shmatikov. "Analysis of Probabilistic Contract Signing". In: *Journal of Computer Security* 14.6 (2006), pp. 561–589.

[113] P. D'Argenio et al. "Reachability analysis of probabilistic systems by successive refinements". In: *PAPM/PROBMIV*. Ed. by L. de Alfaro and S. Gilmore. 2001.

[114] V. Shmatikov. "Probabilistic Model Checking of an Anonymity System". In: *Journal of Computer Security* 12.3/4 (2004), pp. 355–377.

[115] G. Norman et al. "Evaluating the Reliability of NAND Multiplexing with PRISM". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.10 (2005), pp. 1629–1637.

[116] Juan Pablo Vielma. "Mixed Integer Linear Programming Formulation Techniques". In: *SIAM Review* 57.1 (2015), pp. 3–57.

[117] Clark W Barrett et al. "Satisfiability Modulo Theories." In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[118] Pietro Belotti et al. "On handling indicator constraints in mixed integer programming". In: *Computational Optimization and Applications* (2016), pp. 1–22.

[119] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS/ETAPS*. Springer, 2008, pp. 337–340.

# List of Tables

# List of Figures

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Thèse de Doctorat

Anicet BART

## Modélisation et résolution par contraintes de problèmes de vérification

## Constraint Modelling and Solving of some Verification Problems

**Résumé**

La programmation par contraintes offre des langages et des outils permettant de résoudre des problèmes à forte combinatoire et à la complexité élevée tels que ceux qui existent en vérification de programmes. Dans cette thèse nous résolvons deux familles de problèmes de la vérification de programmes. Dans chaque cas de figure nous commençons par une étude formelle du problème avant de proposer des modèles en contraintes puis de réaliser des expérimentations. La première contribution concerne un langage réactif synchrone représentable par une algèbre de diagramme de blocs. Les programmes utilisent des flux infinis et modélisent des systèmes temps réel. Nous proposons un modèle en contraintes muni d'une nouvelle contrainte globale ainsi que ses algorithmes de filtrage inspirés de l'interprétation abstraite. Cette contrainte permet de calculer des sur-approximations des valeurs des flux des diagrammes de blocs. Nous évaluons notre processus de vérification sur le langage FAUST, qui est un langage dédié à la génération de flux audio. La seconde contribution concerne les systèmes probabilistes représentés par des chaînes de Markov à intervalles paramétrés, un formalisme de spécification qui étend les chaînes de Markov. Nous proposons des modèles en contraintes pour vérifier des propriétés qualitatives et quantitatives. Nos modèles dans le cas qualitatif améliorent l'état de l'art tandis que ceux dans le cas quantitatif sont les premiers proposés à ce jour. Nous avons implémenté nos modèles en contraintes en problèmes de programmation linéaire en nombres entiers et en problèmes de satisfaction modulo des théories. Les expériences sont réalisées à partir d'un jeu d'essais de la bibliothèque PRISM.

**Mots clés**

modélisation par contraintes, résolution par contraintes, vérification de programmes, interprétation abstraite, vérification de modèles.

**Abstract**

Constraint programming offers efficient languages and tools for solving combinatorial and computationally hard problems such as the ones proposed in program verification. In this thesis, we tackle two families of program verification problems using constraint programming. In both contexts, we first propose a formal evaluation of our contributions before realizing some experiments. The first contribution is about a synchronous reactive language, represented by a block-diagram algebra. Such programs operate on infinite streams and model real-time processes. We propose a constraint model together with a new global constraint. Our new filtering algorithm is inspired from Abstract Interpretation. It computes over-approximations of the infinite stream values computed by the block-diagrams. We evaluated our verification process on the FAUST language (a language for processing real-time audio streams) and we tested it on examples from the FAUST standard library. The second contribution considers probabilistic processes represented by Parametric Interval Markov Chains, a specification formalism that extends Markov Chains. We propose constraint models for checking qualitative and quantitative reachability properties. Our models for the qualitative case improve the state of the art models, while for the quantitative case our models are the first ones. We implemented and evaluated our verification constraint models as mixed integer linear programs and satisfiability modulo theory programs. Experiments have been realized on a PRISM based benchmark.

**Key Words**

Constraint Modelling, Constraint Solving, Program Verification, Abstract Interpretation, Model Checking.