Sorbonne Université
École Doctorale Informatique, Télécommunications et Électronique

A Combination of Abstract Interpretation and Constraint Programming

Ghiles Ziat

directed by Antoine Miné and Charlotte Truchet

SORBONNE
UNIVERSITÉ

Contents

List of Figures

Chapter 1

Introduction

Contents

The world we live in is getting increasingly influenced by computers. Their performances have improved drastically, as predicted by Moore's laws, whose observation was that the number of transistors in a dense integrated circuit doubles about every two years. Consequently to this growth, and thanks to the huge progress of computer science in the last decades, more and more problems are today being solved using computer aided techniques and the time where Bobby Fischer was discombobulating the Greenblat computer at chess is far behind us. Naturally, the use of computers has not only been standardized in the practice of chess but in a very wide and diverse range of areas, from medicine to air transport through biology and musical composition. This exponential growth of computers' capabilities and usages led to the development of a rich panel of very advanced research fields, and increased the number of sub-domains of computer science. These include, among others, software engineering, computer graphics and visualization, artificial intelligence, computer architecture, theory of computation, quantum information science, and formal methods to name a few.

All these disciplines have seen their development guided by families of very concrete applications. For instance, the increasing popularity of multi-core architectures accelerated the development of concurrent, parallel and distributed systems. In these systems, several computations are executing simultaneously, and potentially interacting with each other. In the field of information transmission, the continual usage of communication and the sharing of sensible information made it necessary to develop and study techniques for secure communications in the presence of malevolent third parties. Hence the emergence of advanced security protocols in cryptanalysis. Also, the introduction of computers in critical embedded systems, such as aircraft autopilot systems or autonomous vehicles made it necessary to have robust methods of software validation to minimize the risk of potentially harmful errors.

In a parallel way, the different fields of computer science have become more and more specialized to answer the specific needs of their different uses, which makes it difficult to have them be aware of each other's progress and share their techniques. Even though the motivations of these fields are very diverse, the techniques they develop to meet their goals share a lot of common ingredients, may this be data, algorithms, languages or architectures. For instance, we can cite the emergence of the General-purpose computing on graphics processing units (GPGPU) which is the use of a graphics processing unit to perform computation in applications traditionally handled by the central processing unit (CPU). GPGPU pipelines were developed for graphics processing purposes as highly parallel units, and were found later on to fit scientific computing needs as well, and have since been developed also in this direction. They are now used in a regular way in bioinformatics, machine learning and molecular dynamics.

Another example of a fruitful collaboration between two research fields is the use of Artificial Intelligence techniques in formal verification, more precisely the use of SAT-SMT solvers in Model Checking [Mon16]: for instance, the Alt-Ergo prover [CIM13, BCC+12] is dedicated to first-order logical formulas coming from program specifications. The theory used in an SMT solver captures specific properties of the program, hence, the link between Artificial Intelligence and Model Checking strongly relies on the definition and management of specific constraint languages. In formal Verification, these specific languages are used to express invariants of a program, in Artificial Intelligence, they are meant to extend the expressivity of the boolean solvers. In practice, the development of modern SMT solvers has really been driven simultaneously by researchers from both fields.

More generally, verification often needs efficient algorithms to compute either the satisfiability of some problems, or some smallest invariant in some language. On the other hand, several Artificial Intelligence techniques have been developed to solve or prove the satisfiability of logical formulas. The notion of tractable constraint languages is at the heart of these two fields. We believe that there are still many possibly successful interactions between Verification and Artificial Intelligence.

1.1 Objective

In this thesis, we exploit the idea of cross-fertilization of computer science fields, and apply it to one member of the formal verification family which is Abstract Interpretation and one area of combinatorial optimization which is Constraint Programming. Both of these fields are interested in the computation of a space of states defined by a system of equations, which corresponds, in one case, to the semantic equations associated to a program, and in the other, to the constraints to be satisfied for a problem. Our objective is to improve Constraint Programming techniques by importing techniques from Abstract Interpretation. More precisely we will abstract core notions of Constraint Programming, construct effective combinations of Constraint Programming and Abstract Interpretation and implement them in a constraint solver. We now present briefly these two fields and explain what interest there is in this hybridization.

1.1.1 Constraint Programming

With the growth of computers capabilities, which makes them the perfect tool for combinatorial exploration of big spaces, it can be tempting to solve combinatorial problems by heavy computations, not to say exhaustive search. But, while a brute-force search is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions. This makes this technique impractical in many cases as the search space tends to grow very quickly as the size of the problem increases. For many real-world applications, they remain too large and an exhaustive search is not tractable enough to be used. Constraint Programming (CP) is a set of techniques that help to speed-up a brute-force algorithm, or to reduce the search space, that is, by using heuristics specific to the problem class. The ultimate goal being that computers solve the problem, we just have to ennunciate it. It is a programming paradigm for declarative description of problems and effective solving of large problems. Constraint programming makes it possible to formalize such problems by using variables to define a certain space and constraints to describe a result to be achieved (accessibility of certain objects for example) within this space. These constraints come with effective algorithms for solving highly combinatorial problems. It is especially useful in areas of planning and scheduling. This field of combinatorial optimization was first introduced in [Mon74] and nowadays the problems solved are getting bigger, both in terms of number of variables and number of constraints, and the constraint solvers are also becoming more complete as they are able to handle more and more kinds of constraints. In 2015, the robot-lab *Philae* woke up on the comet *67P/Churyumov-Gerasimenko* to perform a series of different kinds of signal analysis. These correspond to sequences of activities constrained by two very scarce resources: energy of the main battery and data storage capacity. These experiments were scheduled using Constraint Programming. This field has nowadays a large amount of academic and industrial applications: scheduling and planning problems [GH11, AHP$^+$18], harmonization/composition in music [PRP16], or computer security [LCM18].

1.1.2 Abstract Interpretation

Two other interesting consequences of the growth of computer's capabilities are the fact that programs are nowadays embedded in critical systems, such as planes and missiles, and they are also getting bigger and more complex. For instance, the size of the code embedded in the *Lockheed Martin F-22 Raptor* plane is between one and two million lines of code, and that of a modern car software, a few tens of millions. With the size of the programs, increased the risk of programming errors and the difficulty to find them. The embeding of programs in critical sytems made it necessary to develop formal verification techniques, to validate software, and their important size revealed the need for automated verification techniques, as it is a huge task to perform manually a verification on millions of lines of code. The familly of Automatic static analysis techniques fulfill these needs, and one such being static analysis by Abstract Interpretation. This field was first introduced by *Cousot & Cousot* in [CC77], with solid theoretical bases, to address the problem of program correction, according to a semantic, which is undecidable in the general case. To do that, semantics are mapped to abstract semantics where the properties to prove, are abstracted into less precise properties, but become decidable. Abstract Interpretation techniques allow static analyzers to prove interesting properties on programs, and a particularity of static analysis by Abstract Interpretation is that they are sound by construction. Moreover, the principle of abstraction allows the scaling of analyzing tools for the analysis of very large programs. The static analyzer *Astrée*, for example, succeeded to prove that there were no overflow errors in the electric flight control codes of the Airbus A380 series, which contains about one million lines of code. Today, Abstract Interpretation finds several applications and many analyzers are developed and used in aerospace[SD07], biology [NGVR12], or automotive electronic control systems[YBR$^+$19].

1.1.3 Combining Abstract Interpretation and Constraint Programming

Constraint Programming and Abstract Interpretation are two fields of computer science that manipulate close mathematical concepts. Both are interested in computing a reachable state given an initial state and a set of operations on this state, among which the filtering according to a constraint/guard. When this reachable space is uncomputable, or too costly to compute, they both rely on efficient abstraction techniques to produce an approximation of the space. These similarities in the objectives mean that the techniques developed by one can perhaps be used by the other. Moreeover, the two fields sometimes share the very same concrete problematics, and develop the same solution for it, *e.g.* the algorithm that computes the smallest hull that encompasses a given space was designed independently in Constraint Programming under the name of *HC4* in [BGGP99] and under the name of *bottom-up/top-down* in Abstract Interpretation in [Cou99]. For these reasons, we will be interested during this thesis in the combination of Abstract Interpretation and Constraint Programming.

Previous work. Our work is in line with the work of *Pelleau & al.* on the same subject, which served as a theoretical foundation, as well as a practical basis for our implementation. In particular, we reuse some definitions and results of [Pel12] and [Pel15], in which the author defines abstract domains for Constraint Programming, so as to build a generic solving method. The author also exploits the octagons abstract domain, already defined in Abstract Interpretation to define in it CP-oriented operations, to better take advantage of the octagonal relations expressed in the constraints.

AbSolute. We also develop the abstract solver first presented in [PMTB13a] in which the authors apply techniques from Abstract Interpretation to Constraint Programming. In this paper, the authors highlight some links and differences between these fields regarding fixpoint iterations, and consistencies of Constraint Programming as abstract domains. Finally, the authors introduce a prototype of an abstract constraint solver that leverages abstract interpretation techniques to go beyond classic solvers.

1.2 Contributions

This thesis aims at proposing a tight collaboration between the techniques of Abstract Interpretation and Constraint Programming within a unified method of resolution of constraint satisfaction problems. We will focus on this issue through the following five main chapters:

Chapter 2 (**Preliminaries**) introduces the theoretical preliminaries and replaces our work within the state of the art of both Abstract Interpretation and Constraint Programming. It presents the related works and concludes with a presentation of the previous work on which we base our work, mixing both paradigms, necessary to the good interpretation of our work.

Chapters 3 and 4 focus on a generalization of abstract solver concepts. This generalization allows us to have a more precise handling of constraints and manage more types of variables. In particular:

- in Chapter 3 (**Abstract domains and domain products for Constraint Programming**) , we exploit the reduced product domains of Abstract Interpretation in a framework of constraint resolution. Especially we define a variant of the standard reduced product of Abstract Interpretation, but adapted to Constraint Programming purposes. This gives us a more powerful propagation which results in a gain of time. We use these constructions to address the problem of augmenting the kind of constraints a solver is able to solve, in a generic fashion using domain combinators.

- Chapter 4 (**Discrete and Continuous abstractions for constraint solving**), aims to generalize the variables domains representation, which allows us to mix different types of variables, with different representations. We handle both discrete and continuous variables, by proposing a unified abstraction for the two kinds of spaces, along with the corresponding propagators, splits and size operators.

The next two chapters focus on the design of a more intelligent solving method, in particular for continuous constraints satisfaction problems. The main goal of these chapters is to improve the reusability of the results of a solver, by avoiding unnecessary exploration steps.

- In Chapter 5 (**Propagation with Elimination**) we develop a new technique called *elimination*, whose purpose is to improve the quality of the results of a solver: it reduces the search space by removing from it, as soon as possible, parts that are solutions which leads to better results. Some of the ideas presented in this chapter are already introduced in [ZPTM18]. However, we bring here a deeper attention to some points.

- In Chapter 6 (**Constraint aware Exploration**) we present a new exploration strategy, aware of the constraints of a problem and called *Pizza split*. It defines relevant cutting points within an abstract element and proposes a dedicated exploration strategy. We define pizza split operators for our different abstract domains and use them to tune the solving process.

Lastly, Chapter 7 (**Conclusion**) summarizes our work by returning to all the results obtained during this thesis and discusses the possible continuations and improvements of our work. The various methods mentioned above were implemented in the AbSolute constraints solver which was used for the experimental validation of the ideas described in each chapter. It has also helped us measure the performance of these different techniques. Implementation details, documentation and additional examples are included in the appendix.

CHAPTER 2

PRELIMINARIES

Contents

After we introduce some mathematical background, we present in this chapter the formal definitions of both Abstract Interpretation and Constraint Programming and their related works. We finish by introducing the previous work by *Pelleau & al.* in [Pel12, Pel15, PMTB13a] which is closely related to our work.

2.1 Theoretical Background

Both Constraint Programming and Abstract Interpretation have their theoretical foundations in mathematics. These include notions of arithmetic, set theory, and theory of orders. In this section, we briefly discuss these notions that will be crucial in the rest of this manuscript. We will focus in particular on the notions of lattices and partial orders, fundamental in Abstract Interpretation and at the heart of our hybrid framework, the arithmetic of intervals widely used in both Constraint Programming and Abstract Interpretation, and the concept of fixed points that we use in several cases.

2.1.1 Order Theory

Order theory is a branch of mathematics which investigates the intuitive notion of order using binary relations. It provides a formal framework for describing statements such as "this is less than that" or "this precedes that". Such statements are very useful when it comes to abstracting values and comparing such abstractions. In particular, lattices are the heart of abstract domains which are based on monotonic functions for ordered sets, and thus are central in Abstract Interpretation.

Lattices and Posets

A lattice is an abstract structure studied in the mathematical sub-disciplines of order theory and abstract algebra. It consists of a partially ordered set in which every two elements have a unique supremum (also called a least upper bound or join) and a unique infimum (also called a greatest lower bound or meet). An example is given by the natural numbers, partially ordered by divisibility, for which the unique supremum is the least common multiple and the unique infimum is the greatest common divisor.

Definition 1. *A lattice is a set S equipped with two commutative and associative internal laws, usually noted $\vee$ and $\wedge$ verifying the absorption law.*

$$\forall a, b \in S, a \wedge (a \vee b) = a = a \vee (a \wedge b)$$

The absorption law implies the idempotence of every element of S for the two laws: $a \vee a = a$ and $a \wedge a = a$. For example, Figure 2.1 shows an instance of a finite lattice: the lattice of the subsets of a set $\{a, b, c\}$ ordered by inclusion, with the $\vee$ and the $\wedge$ relations being respectively the usual set-union and the set-intersection.

A lattice is called a complete lattice if all its subsets have both a supremum (join) and an infimum (meet).

From a lattice, we can define an order on S, here noted $\leq$ as a transitive, reflexive and antisymmetric relation:

Definition 2. *An order is a binary relation over a lattice L such that,*

$$\forall a, b \in L, a \leq b \Leftrightarrow a \vee b = b$$

For instance, we can define on the lattice shown in Definition 2.1 an order $a \subseteq b \Leftrightarrow a \vee b = b$ which corresponds to the set-inclusion. Here we can notice that not every pair are comparable, *e.g.* with the

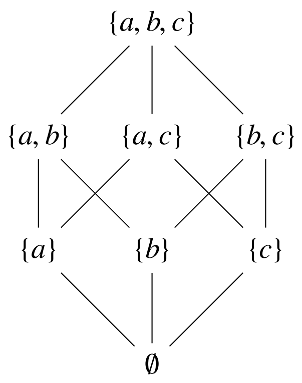


Figure 2.1: Lattice of subsets of $\{a, b, c\}$ ordered by inclusion

pair $\{a, b\}, \{a, c\}$ for which neither $\{a, b\} \subseteq \{a, c\}$ nor $\{a, c\} \subseteq \{a, c\}$ holds, which makes this order not a total order.

A more general concept than a lattice is the notion of partially ordered set A partially ordered set (also called *poset*) formalizes and generalizes the intuitive concept of an ordering, sequencing, or arrangement of the elements of a set. In a poset, not every pair of elements needs to be comparable. That is, there may be pairs of elements for which neither element precedes the other in the poset. Partial orders thus generalize total orders, in which every pair is comparable.

Definition 3. *A Poset $(L, \leq)$ is a set S provided with an order such that,*

$$\forall a \in S, a \leq a \textit{ (reflexivity).}$$
$$\forall a, b \in S, \textit{ if } a \leq b \textit{ and } b \leq a, \textit{ then } a = b \textit{ (antisymmetry).}$$
$$\forall a, b, c \in S, \textit{ if } a \leq b \textit{ and } b \leq c, \textit{ then } a \leq c \textit{ (transitivity).}$$

To be part of a partial order, an order relation must be reflexive (each element is comparable to itself), which is guaranteed by the idempotence, antisymmetric (no two different elements precede each other) which is guaranteed by the commutativity of $\vee$, and transitive (the start of a chain of precedence relations must precede the end of the chain) which we have from the associativity of $\vee$. Also, we can derive a poset from a lattice, as the relation $a \leq b \Leftrightarrow a \vee b = b$ respects Definition 3, but some posets do not correspond to a lattice.

A poset does not necessarily need to be finite and we call infinite poset a poset for which the underlying set is infinite. For example, $\mathbb{N}$ ordered by $\leq$ is an infinite poset.

A more interesting example is the lattice of integer intervals, illustrated in Figure 2.1.1, ordered with inclusion. We denote by $[a, b]$ the closed interval where $a \leq b$, representing the set of elements x satisfying $a \leq x \leq b$.

Also, note that we can have posets that do not form lattices as illustrated by the Hasse diagram in Figure 2.3: here the pair $\{a; b\}$ has no unique least upper bound, but still, we have a partial order induced by the usual visual representation of the diagram stating that if an element x is smaller than another element y, then the point representing x is placed lower than that for y, and there is an edge between the two.

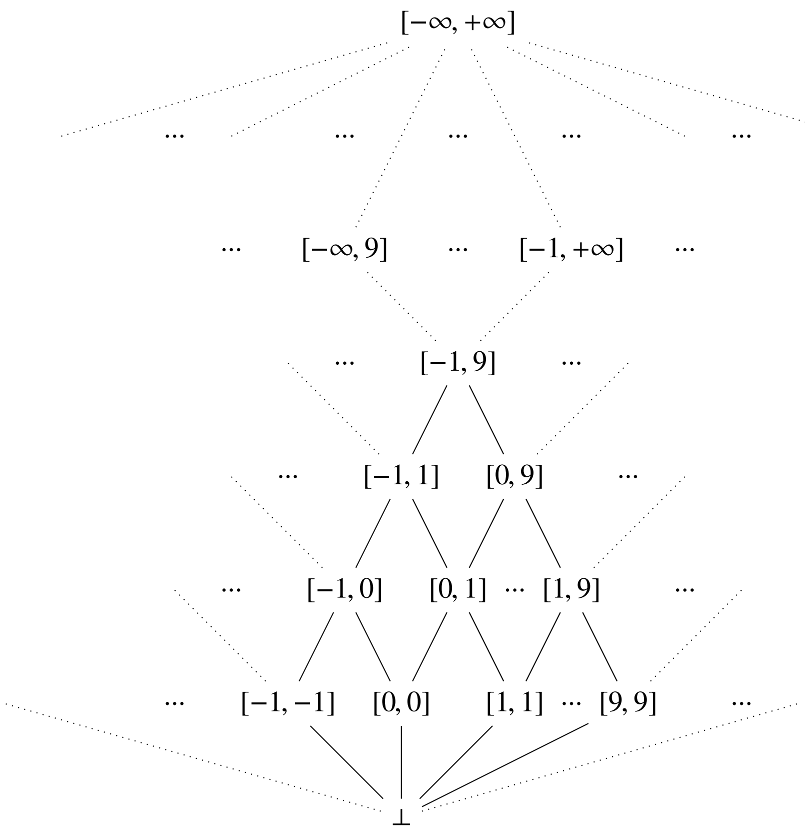


Figure 2.2: Lattice of interval of $\mathbb{Z}$, with $[a, b] \vee [c, d] = [min(a, c), max(b, d)]$ and $[a, b] \wedge [c, d] = [max(a, c), min(b, d)]$

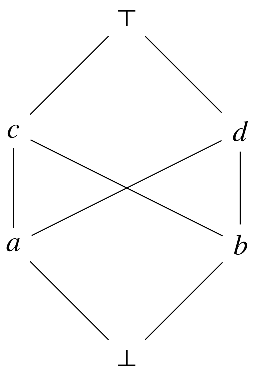


Figure 2.3: Hasse diagram denoting an order

2.1.2 Fixed Points

A fixed point of a function is an element of the function's domain that is mapped to itself by the function. Abstract Interpretation makes use of them to define unbounded aspects of the semantics, such as loops or recursive procedures. This is a central notion that is used to guarantee the termination of an analysis. In Constraint Programming, they are used to define propagation schemes when dealing with several constraints in such a way that in the end, propagation reaches a fixed point and all constraints reach a certain level of consistency.

Definition 4. *A fixed point of a function $f : S \rightarrow S$ is an element $x \in S$ such that $f(x) = x$. A function f may have more than one fixed point and we note* $\mathrm{fp}(f)$ *the set of its fixed points:* $\{x \in S | f(x) = x\}$.

For instance, let a function f be defined on the real numbers by:

$$f(x) = x^3 - 6x^2$$

Then f admits 3 fixed points: $\mathrm{fp}(f) = \{0, 3 + \sqrt{10}, 3 - \sqrt{10}\}$.

Within a poset, we can distinguish the least (respectively greatest) fixpoints which are smaller (respectively bigger) than all other fixed points, according to the subset ordering.

Definition 5. *Given a poset $(S, \leq)$, a least fixed point of a function $f : S \rightarrow S$ is an element $x \in \mathrm{fp}(f)$ such that $\forall y \in \mathrm{fp}(f), x \leq y$*

This definition generalizes naturally to greatest fixpoints. Note that if a least (resp. greatest) fixed point exists, then it is unique (by antisymmetry).

Notations 2.1.1

We note $\mathrm{lfp}(f)$ and $\mathrm{gfp}(f)$ the least fixpoint and the greatest fixpoint of a function f

Also, when a function is defined over an ordered set, it is called a monotonic function if its application preserves or reverses the given order:

Definition 6. *Given a poset $(S, \leq)$, a function $f : S \rightarrow S$ is monotonically increasing if $\forall x, y \in S, x \leq y \Rightarrow f(x) \leq f(y)$*

Which generalizes to monotonically decreasing functions when $\forall x, y \in S, x \leq y \Rightarrow f(y) \leq f(x)$. These definition lead us naturally to the Knaster-Tarski theorem enunciated in Theorem 1.

Theorem 1. *Let $\mathcal{L}$ be a complete lattice and let $f : \mathcal{L} \rightarrow \mathcal{L}$ be a monotonic function. Then the set of fixed points* $\mathrm{fp}(f)$ *in $\mathcal{L}$ is also a non-empty complete lattice.*

From Theorem 1 arises naturally that in particular f admits both a least and a greatest fixpoint. This results is a theoretical mainstay of our work as both Constraint Programming problems and Abstract Interpretation analysis can be seen as the computation of a fixed point of an equation system. The Knaster-Tarski theorem hence ensures that there exists a smallest fixed point for any monotone function and the *Kleene* theorem allows the design of iteration strategies that are guaranteed to find it by iterating the function from the smallest element of the lattice until it reaches the smallest fixpoint, if the iterations terminate.

2.1.3 Interval Arithmetic

Classical arithmetic computations operate over numbers. Interval arithmetic is the reasoning over ranges of numbers. This notion is very practical as intervals are compact in memory, as two integers can represent a potentially very large set of integers, and allow to express exactly classes of invariants that are very useful in practice, which are bounds of variables. Both Abstract Interpretation and Constraint

Programming rely on interval arithmetic for several uses: Interval Abstract Domain, symbolic methods of interval linearization [Min07], expressing non-determinism, or computation with uncertainty. We recall here the definitions of intervals, and introduce interval arithmetic as they will be very useful in the rest of this thesis.

Definition 7. *Given a totally ordered set $(\mathcal{S}, \leq)$, an interval I is a convex subset of $\mathcal{S}$ such that $\forall (a, b) \in I, \forall x \in \mathcal{S}, a \leq x \wedge x \leq b \iff x \in I$.*

Bounds. Bounds are the endpoints of the interval. They entirely define the interval and the interval arithmetic can be derived from an arithmetic over the bounds as we will see.

A closed interval, noted $[a; b]$ is an interval which includes its bounds. An open interval $]a; b[$ is an interval which excludes its bounds. Half-open intervals are intervals with only one of their bounds excluded and are noted $[a; b[$ (respectively $]a; b]$) when the excluded bound is the upper (respectively lower) bound.

In the discrete case, bounds are generally always included in the interval, as removing a point corresponding to a lower (respectively upper) bound either leads to an empty interval or can be expressed as a closed interval as $[a; b[$ can be rewritten as $[a; b - 1]$ when a < b, and as the empty set otherwise.

Discrete intervals. Integer intervals are the sets of closed subset of $\mathbb{Z}$

Continuous intervals. Real intervals are defined as not-necessarily closed connected subsets of $\mathbb{R}$. Thus their endpoints can be either included or excluded.

Extended real (resp. integer) intervals may be defined as a subset of the extended real (integer) numbers, which is the set of all real numbers augmented with $-\infty$ and ∞.

Arithmetic

Interval arithmetic can be built from the interval's bounds arithmetic. For example, from an additive group $(\mathcal{S}, +)$, which is a set equipped with one internal composition law that generalizes the addition, we can build another additive group on the set $\mathcal{S}^2$ that will correspond to the group of the interval of $\mathcal{S}$ provided that we have a total order $\leq$ on $\mathcal{S}$. Hence, given $a, b, c, d \in \mathcal{S}$, with $a \leq b$ and $c \leq d$ we can define the addition operation over intervals as follows:

- $[a, b] + [c, d] = [a + c, b + d]$

Provided that we have the substraction, multiplication and division defined over $\mathcal{S}$, we can also define the other classical arithmetic operations in a similar way:

- $[a, b] - [c, d] = [a - d, b - c]$
- $[a, b] * [c, d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]$

- $[a,b]/[c,d] = [a,b] \cdot \frac{1}{[c,d]}$, where

$$\begin{aligned}
\frac{1}{[c,d]} &= \left[\tfrac{1}{d}, \tfrac{1}{c}\right] && 0 \notin [c,d] \\
\frac{1}{[c,0]} &= \left[-\infty, \tfrac{1}{c}\right] \\
\frac{1}{[0,d]} &= \left[\tfrac{1}{d}, \infty\right] \\
\frac{1}{[c,d]} &= \left[-\infty, \tfrac{1}{c}\right] \vee \left[\tfrac{1}{d}, \infty\right] = [-\infty, \infty] && 0 \in (c,d)
\end{aligned}$$

We will use a lot these kinds of operations as they are a simple way to calculate upper and lower endpoints for the range of values of a function in one or more variables, which will be useful for both Abstract Interpretation and Constraint Programming.

2.2 Constraint Programming

In this section, we first present the main motivations that lead to Constraint Programming, we then introduce a few domains related to it. We finally give the formal definitions of Constraint Programming and some examples.

2.2.1 Motivation

Constraint programming techniques are widely used in the industrial world in the field of decision support. Decision support is the set of techniques allowing, for a given person, to opt for the best possible decision. Many works deal with scheduling and project management, but also logistics (vehicle tours, packaging ...), planning, and scheduling problems. In the context of the manufacturing industry, decision support procedures make it possible to find production plans (production scheduling), to have the machines of a workshop run in the best possible way, to reduce the waste of raw materials (cutting problems) or energy consumption or to optimize the packaging and delivery of intermediate or finished products. In the field of finance, investment problems are classic problems of optimization. They generally consist in maximizing the profit (or the profit expectancy) obtained from a given amount by combining at best the different possibilities offered to the investor. The applications in the field of computing are very numerous too. We can cite, among other things, the choice of the location and number of servers to put in place, the storage capacity, the computing power and the network throughput, the choice of a computer architecture (centralized application / distributed, real-time or delayed processing, mesh or star network, etc.), and in operating systems and program verification [MRL01].

Operations Research

Operations research can be defined as the set of rational methods and analytic techniques oriented towards finding the best choice in how to operate in order to achieve the desired result or the best possible result. It is part of the "decision aids" in that it proposes conceptual models to analyze and master complex situations to enable decision-makers to understand, assess issues and arbitrate or to make the most effective choices. Among approaches used in operations research are mathematical logic, simulation, network analysis, queuing theory, and game theory. It is a field of feasibility analysis based on a solution-driven approach: from a set of potential solutions to a problem, an analysis reduces it to a small set of solutions most likely to be usable. The alternatives derived are then subjected to simulated implementation and, if possible, tested out in real-world situations.

Boolean Satisfiability Problem

Boolean satisfiability problem, also called SAT problem, is a family of problems where the goal is to determine if there exists an interpretation that satisfies a given Boolean formula. It is a problem whose origins go back to [DP60] and later on in [DLL62] with the Davis–Putnam–Logemann–Loveland (DPLL) algorithm which is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas. SAT problems ask whether the variables of a given Boolean formula can be consistently replaced by the values *true* or *false* in such a way that the formula evaluates to *true*. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is *false* for all possible variable assignments and the formula is unsatisfiable. For example, the formula $a \wedge \neg b$ is satisfiable because one can find the values $a = true$ and $b = false$, which makes $a \wedge \neg b = true$. On the opposite, $a \wedge \neg a$ is unsatisfiable.

Satisfiability Modulo Theories

Another application that often involves the DPLL algorithm lies in the field of satisfiability modulo theories (SMT). This is a SAT problem in which propositional variables are replaced with formulas of another logical theory. It relies on combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays[Nel80], bit vectors and so on. SMT can be thought of as a form of constraint satisfaction problem and thus a certain formalized approach to Constraint Programming.

Constraint Programming

Constraint Programming is a resolution framework, which was first introduced in [Mon74]. It is a paradigm of declarative programming in which a user states the mathematical laws that govern a system, and then relies on a solver to find the satisfying instances of his specification.

2.2.2 Introductory Example

Lets consider a concrete example: The Minesweeper game. The Minesweeper is a puzzle game whose purpose is to locate mines hidden in a virtual filed which is a 2D grid with the only indication being the number of mines in adjacent areas for each square as shown in Figure 2.4. The game can be seen as a constraint satisfaction problem, where each square is a variable with value in $\{0; 1\}$ (denoting respectively the absence or the presence of a mine on the given square).

The game gives visual information on the visible squares, that can be seen as constraints as they restrict the possible values of some of the squares.

- The number inside a square indicates the number of mines in its neighborhood

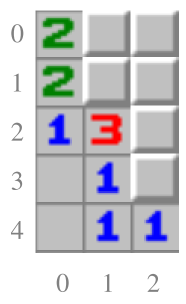


Figure 2.4: An example of a game of Minesweeper

- On the visible squares, no mine is present

On the game shown in Figure 2.4, if we note $X_{(i,j)}$ the square of coordinates (i, j) with $j \in \{0, 1, 2, 3, 4\}$ and $i \in \{0, 1, 2\}$, we can write down the corresponding constraint system, where the constraints are to be understood conjunctively, as follows:

- $X_{(0,1)} + X_{(1,1)} + X_{(1,0)} = 2$ (from square (0,0))
- $X_{(0,0)} + X_{(1,1)} + X_{(1,0)} + X_{(0,2)} + X_{(1,2)} = 2$ (from square (0,1))
- $X_{(0,1)} + X_{(1,1)} + X_{(1,2)} + X_{(0,3)} + X_{(1,3)} = 1$ (from square (0,2))
- $X_{(0,1)} + X_{(1,1)} + X_{(2,1)} + X_{(0,2)} + X_{(2,2)} + X_{(0,3)} + X_{(1,3)} + X_{(2,3)} = 3$ (from square (1,2))
- $X_{(0,2)} + X_{(1,2)} + X_{(2,2)} + X_{(0,3)} + X_{(2,3)} + X_{(0,4)} + X_{(1,4)} + X_{(2,4)} = 1$ (from square (1,3))
- $X_{(0,3)} + X_{(1,3)} + X_{(2,3)} + X_{(0,4)} + X_{(2,4)} = 1$ (from square (1,4))
- $X_{(1,3)} + X_{(2,3)} + X_{(1,4)} = 1$ (from square (2,4))

Which after simplification from the visible squares $X_{(0,0)} = X_{(0,1)} = X_{(0,2)} = X_{(1,2)} = X_{(0,3)} = X_{(1,3)} = X_{(0,4)} = X_{(1,4)} = X_{(2,4)} = 0$, and removal of redundant constraints, becomes:

- $X_{(1,0)} + X_{(1,1)} = 2$ (from square (0,0) and (0,1))
- $X_{(1,1)} = 1$ (from square (0,2))
- $X_{(1,1)} + X_{(2,1)} + X_{(2,2)} + X_{(2,3)} = 3$ (from square (1,2))
- $X_{(2,2)} + X_{(2,3)} = 1$ (from square (1,3))
- $X_{(2,3)} = 1$ (from square (1,4) and (2,4))

From this simplified constraint system, we are able to determine directly that $X_{(2,3)}$ and $X_{(1,1)}$ contain a mine, and so does $X_{(1,0)}$ as the square $X_{(0,0)}$ has only two possible neighbours that may contain a mine, and we know that it has two mines in its neighborhood, $X_{(1,0)}$ and $X_{(1,1)}$. Figure 2.5 shows the state of the game after these deductions have been performed.

Applying the same reasoning one more step allows us to discard the square $X_{(2,2)}$ from the possible mined squares and to deduce that $X_{(2,1)}$ does contain a mine. From this point, illustrated by Figure 2.6, no more deduction can be made. The square $X_{(0,2)}$ being possibly both a mine or an empty square.

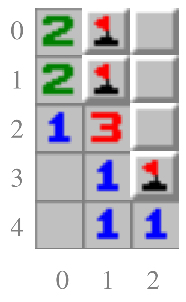


Figure 2.5: The same game after a few step of reasoning

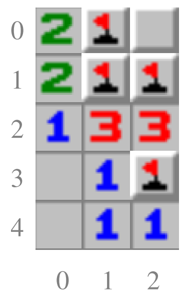


Figure 2.6: Final state of the game from which we cannot perform anymore deduction

The resolution process we have just made explicit on this example is a typical form of Constraint Programming: from a concrete application, we have modelled it in a mathematical way, using variables, domains of possible values for these variables, and constraints over these variables. From this modelling, we exploited the constraints both to discard impossible states of the game, and to infer the only possible values of some variables, until we have reached a consistent system on which no more refinement can be made.

Also, the example we have just presented features discrete variables with only two possible values. A more realistic kind of problems, which is closer to the ones we will handle in this thesis, is the continuous problems, where the variables generally have an infinite number of possible values, even though their ranges are bounded.

One such is the testing of complex industrial robots [MGM15], that involve several interacting control systems. These industrial robots are complex cyberphysical systems which require to be rigorously tested before being deployed. In this context, the goal is to make sure that a robot's trajectory, under some configuration, should stay inside a given space.

A robot has a complex axis-motion system. These axes can have several degrees of freedom and define an accessible area by the arm of the robot. The reachable space by the robot is divided into two distinct sub-spaces. The first one is a prohibited zone, which is an area in which the robot is not allowed to go to (because of constraints foreign to the robot: objects, humans, etc. that make the movement of the robot potentially harmful). The remaining space is the work area in which the robot is allowed to go. A valid trajectory then consists of segments that cross the work area without ever crossing the prohibited zone. The modelling of the space configuration is based on basic geometric shapes (lines, squares, triangles, circles ...) which can be seen as a constraint system defining the reachable space by the arm of the robot but for the sake of coherence with the other problems we will illustrate here, we give the constraints in an equational form.

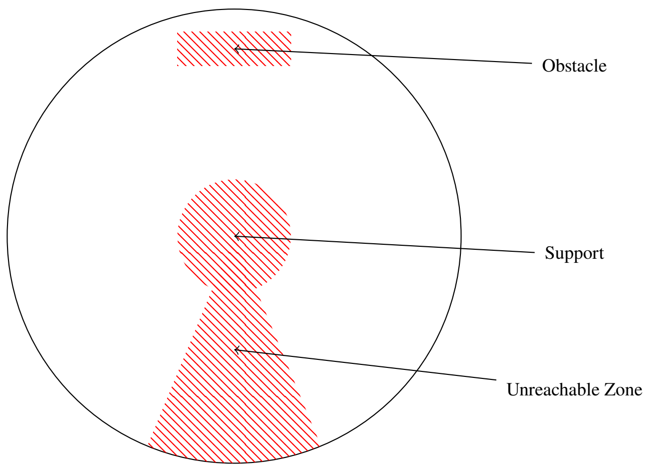


Figure 2.7: Working zone and Forbidden zone of a Single-Arm robot

$$x^2 + y^2 \leq 2 \tag{2.1}$$

$$2 * x > -y \lor 2 * x < y \tag{2.2}$$

$$x^2 + y^2 \geq 0.5 \tag{2.3}$$

$$x > 0.5 \lor x < -0.5 \lor y > 1.8 \lor y < 1.5 \tag{2.4}$$

The arm of the robot allows it to reach every point within the interior of a disk (Equation 2.1), from which we remove a sector that is unreachable due to a mechanical constraints of the robot's arm (Equation 2.2). We also remove the space corresponding to the foot support of the robot itself which is also a disk (Equation 2.3). Finally, a last constraint is due to some obstacle placed around the robot and where the robot's arm cannot move without provoking a dangerous collision (Equation 2.4). Figure 2.7 gives a visual representation of the space defined by these consraints, where the red hatched part is the forbidden area.

Solving this kind of problems differs from solving a discrete problem with variables having finite domains. Yet the goal is still the same: we want to avoid a naive exhaustive search and exploit the structure of the problem to find solutions more effectively. State of the art solvers in Constraint Programming handle this kind of problems by producing a coverage of the solution space with boxes. This representation makes it possible to compute its intersection with other spaces, or to approximate its volume and so on.

In the present work we will develop a technique inspired from continuous solvers of Constraint Programming, that is able not only to produce a coverage of the solution space with boxes, but with any specific shape that respects some condition as we will see later on.

2.2.3 Definitions

We now give the mathematical definitions for Constraint Programming which we illustrate with several examples.

Constraint programming is a paradigm of declarative programming, which means that its goal is to answer the question *"What is the problem?"* counter to imperative programming which describes *"How* *to solve the problem?”*. To do that, we give the description of problems in mathematical terms involving constraints over variables, and rely on a solver to get the solution.

A variable is a quantity whose value is indeterminate, but on which a combination of operations is performed. A variable is generally not fully specified, its value may even be completely unknown, but it must belong to a set which is called its domain.

A constraint is a relationship between several variables that limits the set of values that these variables can take simultaneously. It is a mandatory condition that must be satisfied by the solution of a problem. Constraints can be arithmetic (equalities, inequalities) or more complex (global constraints), and can involve one or several variables.

Definition 8. *In Constraint Programming, constraint satisfaction problems (CSP) are modeled using triplets* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*, where n and m are respectively the number of variables and the number of constraints of the problem.:*

- $\mathcal{X} = \{x_1, ..., x_n\}$*, the variables of the problem*
- $\mathcal{D} = \{d_1, ..., d_n\}$*, the domains of the variables such that* $x_k \in d_k, \forall k \in [1, n]$
- $\mathcal{C} = \{c_1, ..., c_m\}$*, the constraints of the problem*

We call an instance (or assignment) a total mapping $\mathcal{X} \to \mathcal{D}$. A mapping is said to be partial if not all the variables are mapped to a value. The set of all the assignments is called the search-space. A solution of a CSP is an instance that satisfies all of the constraints.

Definition 9. *A solution of a CSP* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is an instance* $i = \{x_1 \to v_1, ..., x_n \to v_n\}$ *that verifies all of the constraints by substitution of the variables with their value in i.*

$$\forall c \in \mathcal{C}, c\{x_1 \to v_1, ..., x_n \to v_n\}$$

Notations 2.2.1

We use thereafter the following notations to designate the instances that are solutions of a CSP and the ones that are not. We call the latter the inconsistent instances.

- $Sol(<\mathcal{X}, \mathcal{D}, \mathcal{C}>)$ the set of all the solutions of a CSP.
- $\overline{Sol}(<\mathcal{X}, \mathcal{D}, \mathcal{C}>) = \{x : \mathcal{X} \to \mathcal{D} | x \notin Sol(<\mathcal{X}, \mathcal{D}, \mathcal{C}>)\}$ to designate the set of inconsistent values that belong to the search-space.

From this framework, problems and solutions are well defined, but nothing determines how to find the solutions associated to a problem, and how to choose between solutions. We will differentiate two very distinct forms of constraint satisfaction problems: given a problem p, one can wish to find all of its solutions $Sol(p)$ or find a single solution $s \in Sol(p)$.

We now present some of the most used techniques to deal with these two problems.

2.2.4 Constraint Solving Methods

Theoretically, when the search space is finite, CSP solving is an easy task. A simple exhaustive enumeration of the search space, filtered by the constraints, is enough to build the solution space. This is the *generate-and-test* procedure. The idea is simple: a complete labeling of the variables is generated and, therefore, if this labeling satisfies all the constraints, then the solution is found, otherwise another labeling is generated, and the process may be repeated until all of the search space has either been discriminated as a solution or an inconsistent value. Algorithm 1 sketches this procedure, using the following auxiliary procedures:

- generate(P): generates the next candidate solution for P. It returns $NULL$ when all candidates have been enumerated
- test(P,candidate) checks whether candidate is a solution for P.
- output (P, candidate): uses the solution candidate of P as appropriate to the application.

Algorithm 1 Generate-and-Test procedure

```
function GENNTEST(P)                                        ▷ P: a problem
    candidate ← generate(P)
    while candidate $\neq NULL$ do
        if test(P,candidate) then
            output(P, candidate)
        else
            c ← generate(P,candidate)
```

Of course, this procedure suffers from many limitations when the search space is infinite, or simply very large. That is why several improvements have been made over this algorithm like symmetry breaking as in [GPfP06] or [Wal06] which can be applied in some cases when the problem does feature a symmetry (*e.g.* eight queens puzzle), or more often, search space reduction with principles *à la* minimax as in [Liu98] or [HK14] which is based on an evaluation function which provides no guarantee of being optimal but which can be very useful in large-scale applications. Both of these techniques are orthogonal to our work which focuses on constraint propagation to reduce the search space.

2.2.5 Propagation

Constraint propagation works by reducing the domains of the variables, strengthening the constraints, or creating new ones. This leads to a reduction of the search space, making the problem easier to solve by some algorithms. The notion of propagation links a constraint to its actual implementation through propagators. Propagators are functions representing a way to infer that a constraint forbids some values from the domains.

Definition 10. *Given a CSP $< \mathcal{X}, \mathcal{D}, C >$, a function $\rho : P(\mathcal{D}) \rightarrow P(\mathcal{D})$ is a propagator for a constraint $c \in C$ if and only if:*

- $\forall d \in \mathcal{D}, \rho(d) \subseteq d$ *(contraction)*
- $\forall s \in Sol(< \mathcal{X}, \mathcal{D}, \{c\} >), s \in d \implies s \in \rho(d)$ *(completeness)*

The contraction property guarantees that a propagator can only reduce the domains of the variables and the soundness guarantees that no solution is removed from a domain. In the Abstract Interpretation terminology, $\rho(c)$ is an sound over-approximation of $Sol(<\mathcal{X},\mathcal{D},\{c\}>)$. Note that there may exist several propagators, with different cost/precision trade-offs, for a single constraint (*e.g.* the identity function is a valid propagator for any constraint according to our definition).

When a propagator reduces the search space in such a way that it can not reduce it more without losing solutions (*i.e.* breaking the completeness property), we say that the obtained assignment (which may be partial) is consistent. The "standard" local consistency conditions all require that all consistent partial instantiations can be extended to another variable in such a way that the resulting assignment is consistent. A partial instantiation is consistent if it satisfies all the constraints which involve a subset of the assigned variables.

The notion of *local consistency* is central in Constraint Programming. Local consistency conditions are properties of constraint satisfaction problems related to the feasibility of a problem with respect to some subsets of its variables or constraints. These properties give some extra information about the problem that can be used by the solving process. Various kinds of local consistency conditions are leveraged among which node-consistency, arc-consistency, and path-consistency etc.

Definition 11. *Given $\mathcal{X}$ a set of variables and $\mathcal{D}$ their domains, a possibly partial assignments $i \in \mathcal{X}' \to \mathcal{D}$ with $\mathcal{X}' \subseteq \mathcal{X}$ is said to be locally consistent with respect to a constraint c if the following condition holds:*

$$\exists j \in Sol(\mathcal{X},\mathcal{D},\{c\}), \forall x \in \mathcal{X}', i(x) = j(x)$$

In other words, local consistency ensures that all consistent partial assignments can be extended to another variable without violating the constraint.

For example, given two variables $v_1 \in \{1,2,3,4\}$ and $v_2 \in \{3,4,5,6\}$ and a constraint $c = v_1 \geq v_2$, the partial assignment $\{v_1 \to 1\}$ is inconsistent as v_2 can take no value that would satisfy c, and for symmetrical reasons, the partial assignments $\{v_1 \to 3\}$ and $\{v_1 \to 4\}$ are locally consistent. This definition can be strengthened with several conditions to obtain different forms of consistency and we present here few of them.

Arc-consistency. A variable is arc-consistent with another one according to a binary constraint c if each of its admissible values is consistent with some admissible value of the second variable. Formally, a variable x_i is arc-consistent with another variable x_j if, for every value v_i in the domain of x_i there exists a value v_j in the domain of x_j such that (v_i, v_j) satisfies the binary constraint c. A problem is arc-consistent if every variable is arc-consistent with every other one.

Path-consistency. Path-consistency is a property that generalizes the idea of arc-consistency. It does so by not considering pairs of variables but triplets. A pair of variables is path-consistent with a third variable if each consistent evaluation of the pair can be extended to the other variable in such a way that all binary constraints are satisfied. Formally, x_i and x_j are path consistent with x_k if, for every pair of values (a, b) that satisfies the binary constraint between x_i and x_j, there exists a value c in the domain of x_k such that (a, c) and (b, c) satisfy the constraints between x_i and x_k and between x_j and x_k, respectively.

Bound-consistency. This consistency is based on the consistency of the extreme values of the domains, that is, the minimum and maximum values a variable can take. It denotes that, for a given constraint ctr, there exists for every variable v_i at least one solution to ctr such that $v_i = \underline{v_i}$, and every other variable is assigned to a value located in its range, and there exists at least one solution to ctr such that $v_i = \overline{v_i}$ and every other domain variable is assigned to a value located in its range, with $\underline{v_i}$ and $\overline{v_i}$ being respectively the minimum and maximum values the variable v_i can take.

Consistencies give hint about the satisfiability of a problem. Consistency properties generally ensure that the domains are an over-approximation of the feasible set. This implies that when a propagator reaches consistency, it might produce an empty domain or an unsatisfiable constraint. In this case, the problem has no solution. Being able to prove that a problem is locally-consistent is very useful for a solving process: for example, enforcing arc consistency establishes satisfiability of problems made of binary constraints with no cycles (*e.g.* a tree of binary constraints).

Consistencies for Continuous constraint solving. In the case of continuous constraint solving, other very useful consistencies are used to highlight crucial characteristics of a problem. For instance, instead of reasoning on finite sets of values, we generally require an encoding that is able to express infinite sets of values. We recall the definition of Hull-consistency [BGGP99], one of the classic local consistencies for continuous constraints.

Let $\mathbb{R}$ be the set of reals extended with the infinities $-\infty, \infty$, and $\mathbb{F} \subseteq \mathbb{R}$ a finite subset of reals corresponding to binary floating-point numbers. For practical reasons, we neglect the processor precision (whether 32 or 64 bits) as it does not influence on the techniques we use.

A closed/open floating-point interval is a connected set of reals whose lowest upper bound and greatest lower bound are floating-point numbers.

Notations 2.2.2

The following notations are used as shorthand to designate closed, half-open, and open intervals, where $l \in \mathbb{F}$ and $h \in \mathbb{F}$.

- $[l; h] \triangleq \{r \in \mathbb{R} | l \leq r \leq h\}$,
- $[l; h[\triangleq \{r \in \mathbb{R} | l \leq r < h\}$,
- $]l; h] \triangleq \{r \in \mathbb{R} | l < r \leq h\}$,
- $]l; h[\triangleq \{r \in \mathbb{R} | l < r < h\}$,

Lifting up the interval concept to several dimensions gives us the notion of Box. It is a compact, not necessarily closed, convex figure.

Definition 12. *Let $\mathcal{B}$ be the set of Cartesian products of n intervals*

$$\mathcal{B} = i_1 \times \cdots i_n \triangleq \{r_1 \times \ldots \cdots r_n \in \mathbb{R}^n | r_1 \in i_1 \wedge \cdots r_n \in i_n\}$$

Given a box, one can wish to remove from it the inconsistent instantiations. However, discarding all values of a box for which a real constraint does not hold is not achievable in general, hence a notion of consistency is defined for boxes, which is called the hull consistency.

Definition 13 (Hull-Consistency)**.** *Let $\mathcal{X} = x_1, \ldots, x_n$ be variables over continuous domains represented by the box $\mathcal{D} = d_1, \ldots, d_n$, and c a constraint. The domains are said to be Hull-consistent for c if and only if:*

$$\forall \mathcal{D}' \in \mathbb{B}, Sol(< \mathcal{X}, \mathcal{D}, \{c\} >) \subseteq \mathcal{D}' \implies \mathcal{D} \subseteq \mathcal{D}'$$

In other words, the domains are Hull-consistent if and only if the box $\mathcal{D} = d_1 \times \cdots \times d_n$ is the smallest floating-point box containing the solutions for the constraint c, and this smallest floating-point box always exists as $\mathbb{B}$ is finite and closed by intersection.

The domains which are locally consistent for all constraints are the largest common fixpoints of all the constraint propagators [Ben96, SS08]. In practice, propagators often compute over-approximations of the locally consistent domains. In the following, we will use the standard algorithm HC4 [BGGP99], which propagates continuous constraints, relying on the syntax of the constraints and interval arithmetic [Moo66], although our method could be combined with other propagators. HC4 generally does not reach Hull consistency in a single iteration, in particular in case of multiple occurrences of the variables in the constraints.

Local consistency computations can be seen as deductions, performed on domains by analyzing the constraints. If the propagators return the empty set, the domains are inconsistent and the problem has no solution. Otherwise, non-empty local consistent domains are computed. This may be not sufficient to provide an accurate characterization of the solution set which may not be represented exactly by its bounding box. In that case, choices are made on the variables values. For continuous constraints, typically a domain d is chosen and split into two (or more) parts, which are in turn narrowed by the propagators. The solver alternates propagation and split phases until a given precision is reached, *i.e.* all the boxes which are still considered are smaller than a given parameter. Of course, as soon as a box is proven to contain only solutions, it can be removed from the search space and added to the solution set. Upon termination, the collection of boxes returned covers the solution set $\mathcal{S}$, under some hypotheses on the propagators and splits [Ben96].

Exploration

Propagation alone does not make it possible to instantiate all the variables, and it is therefore necessary to proceed to a supplementary solving step. On such idea is the exploration technique. It consists in splitting the problem into several sub-problems (for example by instantiating a variable at each of its possible values) and to restart filtering on each of these parts, the idea being that the obtained sub-problems should be easier to solve.

Definition 14. *Given a CSP $p =< \mathcal{X}, \mathcal{D}, \mathcal{C} >$, a function* explore $: < \mathcal{X}, \mathcal{D}, \mathcal{C} > \rightarrow \mathcal{P}(< \mathcal{X}, \mathcal{D}, \mathcal{C} >)$ *is an exploration function if and only if it respects the following properties: Let $p'_0, \ldots, p'_n =$* explore(p) *be the sub-problems obtained by exploration of the problem p,*

$$\forall s \in Sol(p), s \in \bigcup_{i=0}^{n} Sol(p'_i) \textit{ (injection)}$$
$$\forall i \in [0, n], Sol(p'_i) \subseteq Sol(p) \textit{ (surjection)}$$

The injection property ensures that no solution is lost and the surjection property guarantees that no solution is created by the exploration process.

Algorithm 2 Propagation - Exploration loop

```
1: function SOLVE($P$:problem)
2:     if $\neg$stop($P$) then
3:         $P' \leftarrow$ propagate($P$)
4:         for $P'' \in$ explore($P'$) do
5:             solve($P''$)
```

This step usually consists in building sub-problems by dividing the domains of the variables into sub-domains or by adding constraints in each sub-problem in such a way that the added constraints are complementary. This keeps the set of solutions the same as for every constraint c, any element of $Sol(p)$ satisfies either c or its complement.

We can strengthen Definition 14 to enforce a non-redundancy property, that is, no instance is a solution of more than one sub-problem.

Definition 15. *An explore function f is said to be irredundant if it is an explore function according to Definition 14 and it satisfies the following property:*

$$\forall s \in Sol(p'_i), s \in Sol(p'_j) \implies i = j \text{ (irredundancy)}$$

Even though this property is not mandatory to build a sound and complete solving process, it becomes very handy from a practical point of view.

2.2.6 Effective Solving

Propagation tries to reduce the search space and exploration divides problems into smaller sub-problems. Repeating these two steps in alternation is almost sufficient to build a complete solving process, the last thing we require is to enforce termination. To do this, we can simply limit the total number of steps of the procedure, or the maximum depth of computation. One can also continue the solving until consistency is reached, if guaranteed to be reached or use a more problem-aware technique that measures properties on the current sub-problem and decides accordingly if the solving must continue or not.

Algorithm 2 illustrates this procedure. Note that this sketch of a resolution procedure does not depend on the exploration heuristic nor the propagator used. We rely on this property to develop a generic solving method.

2.2.7 Discrete & Continuous

A continuous variable is a variable whose set of values it can take is uncountable. For example, a variable over a non-empty range of the real numbers is continuous, if it can take any value in that range. In contrast, a discrete variable over a particular range of real values is one for which, for any value in the range that the variable is permitted to take on, there is a positive minimum distance to the nearest other permissible value. The number of permitted values is either finite (*e.g.* finite sets of values) or infinite but countable (*e.g.* integers, non-negative integers, positive integers).

Discrete

In the discrete case, a straightforward characterization of the solution space is an enumeration of the instances that belong to it. Also, in discrete constraints, domain labels are generally represented simply as enumerations of values or value combinations. Example 1 and Figure 2.8 show[1] respectively an example of a discrete CSP and its solution set.

Example 1. *A constraint system with two discrete variables and two non-linear constraints:*

- $V = (v_1, v_2) \in \mathbb{Z}$
- $D_1 = [-1, 14], D_2 = [-5, 10]$
- $C_1 : (v_1 - 9)^2 + v_2^2 \leq 25$
- $C_2 : (v_1 + 1)^2 + (v_2 - 5)^2 \leq 100$

Using a simple procedure like Algorithm 1 is enough to obtain the set of solutions of this problem, but it may be interesting to use some more advanced strategies when the domains of the variable grow larger.

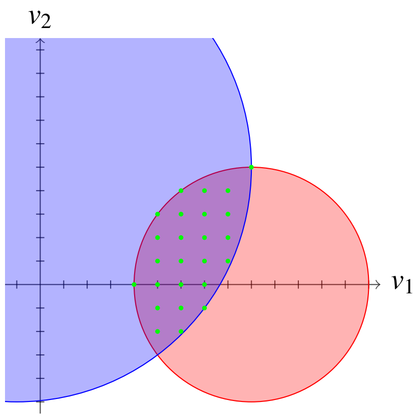


Figure 2.8: The 25 solutions (in green) of the constraint system shown in example 1

Continuous

In the continuous case, on the other hand, finding a characterization of the solution space becomes harder. One common representation is the box partitioning of the solution space. Here, each domain is represented by one or a small collection of intervals. Example 2 and Figure 2.9 show respectively an example of a discrete CSP and its solution set.

Example 2. *The same constraint system as in Example 1, only with continuous variables now.*

- $V = (v_1, v_2) \in \mathbb{R}$
- $D_1 = [-1, 14], D_2 = [-5, 10]$
- $C_1 : (v_1 - 9)^2 + v_2^2 \leq 25$
- $C_2 : (v_1 + 1)^2 + (v_2 - 5)^2 \leq 100$

[1]For the sake of clarity and simplicity, most of the figures inside this document will be in 2D, illustrating the solving of two-variables problems, but the methods we propose tackle just as well higher dimensions spaces.

The solving of a continuous constraint satisfaction problem can consist in paving it with an easy to understand representation, here boxes. Even though the partitioning obtained is less precise than the constraint description of the solution space, it is still easier to use from a practical point of view. For instance it is easier to approximate its volume, or to generate randomly and quasi-uniformly an instance in the solution space.

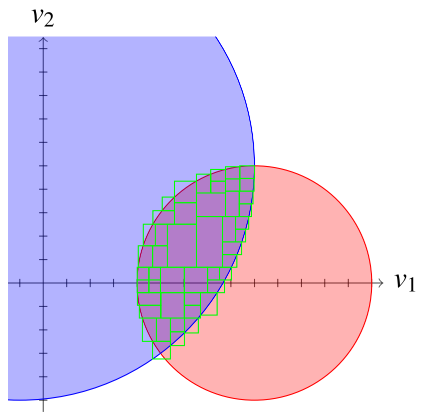


Figure 2.9: The 48 box solutions (in green) of the constraint system shown in example 2

2.3 Abstract Interpretation

In this section, we first present the main motivations that lead to Abstract Interpretation, we then introduce a few domains related to it. We finally give the formal definitions of an analysis by Abstract Interpretation and some examples.

2.3.1 Motivation: Program Verification

Software errors (or "bugs") can cause computer systems to malfunction, with consequences ranging from mere annoyance to large economical losses. Measuring the economical impact of a software error is not a trivial task [SSY15], but this impact finds its culmination in critical embedded systems. In those cases, even simple programming errors can cause the loss of human lives. For example, a famous automobile firm was convicted by a court for a case of unintended acceleration that lead to the death of one of the occupants. Central to the trial was the Engine Control Module's firmware[Cum16]. Hence, program verification to ensure that programs obey their specifications and are free from run-time errors, is a significant part of program development, even accounting to more than half of the development cost for critical systems. As programs are getting larger and more complex, and verification costs rise, it is critical to employ more efficient verification methods, and in particular use automated techniques that can leverage the power of computers. Among the most common and easily deployable verification techniques is software testing. Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test meets the requirements that guided its design and development, responds correctly to all kinds of inputs, performs its functions within an acceptable time, is sufficiently usable, can be installed and run in its intended environments, and achieves the general result its stakeholders desire.

Limitations. Testing methods are able to find bugs, but can not guarantee their absence as they can not be exhaustive in case of too large or infinite inputs.

2.3.2 Formal Methods

Due to the limitations of testing, another family of program verification techniques has appeared which is formal methods. Formal methods are a particular kind of mathematically based technique for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. We now present some of the most widespread formal methods.

Model Checking

Model checking is a verification framework of finite state systems introduced in the early 1980's by [QS82] and [CE81]. Given a model of a system, and its specification, it consists in exhaustively and automatically checking whether this model meets the given specification. Typically, one has hardware or software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems. In order to solve such a problem in an algorithmic way, both the model of the system and the specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. This general concept applies to many kinds of logic and suitable structures. A simple model checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure. One of the characteristics of model checking is the use of rather powerful specification languages, based on temporal logic [Bou09].

Limitations. Exhaustiveness narrow the use of Model Checking to relatively small models, either finite or regular. So we rarely check the actual program, but rather a very simplified model, written by hand for the purposes of verification.

Proof Assistants

A proof assistant is a tool to assist with the development of formal proofs by human-machine collaboration. This involves some sort of interactive proof editor, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer. For example *Coq*[Ler09] allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to find formal proofs, and extracts a certified program from the constructive proof of its formal specification.

Coq works within the theory of the calculus of inductive constructions, a derivative of the calculus of constructions. Coq is not an automated theorem prover but includes automatic theorem proving tactics and various decision procedures. An example application of Coq and its support for certified program extraction is *CompCert*, a formally verified optimizing compiler for a large subset of the C99 programming language [KLW14].

Limitations. As a proof assistant requires manpower and is not an automated tool, it is well adapted to a design of software which is done at the same time as the proof of correction, hand in hand, but less to a verification of the entirety of a program already written. Also, the guarantees it can provide are still limited to what the user wants/manages to prove, and thus still subject to human omission.

Static Analysis

In a more automatic way, static analysis allows gathering knowledge about the runtime behaviour of a program without even running it. There are several forms of static analysis which share these common characteristics: these analyzes always terminate, are automatic as they don't require any action from the user, they rely on approximations to bypass problems of undecidability and to ensure efficiency, and they are sound as they consider over-approximations of all behaviors.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. It can also be useful for the purpose of program optimization. If a compiler can prove that a program is safe, then it does not need to emit dynamic safety checks, allowing the resulting compiled binary to run faster and to be smaller.

One well-known example of static analysis is type checking. Static type checking is the process of verifying the type safety of a program based on analysis of a program's text. If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs. Static type checking can be considered a limited form of program verification.

Static type checking is rather conservative as it rejects programs that may be valid but which are hard to prove correct.

```
let x = 2 in
(if x mod 2 = 0 then 5 else "five") + 1
```

Listing 2.1: A correct program rejected by type checking

For instance, Program 2.1 fails to pass the type check (in most statically typed languages, here **OCaml**), because both branches of the conditional do not have the same type. This is made necessary by the type checker as the value of the condition **(x mod 2)** cannot be statically determined, even though in this case it is true and the program would not produce an error if it ran.

Type safety contributes to program correctness, but might only guarantee correctness for a certain kind of errors that are prohibited by the type system. In a type system with automated type checking a program may prove to run incorrectly yet be safely typed, and produce no compiler errors.

For instance, Program 2.2 illustrates a division by zero which is an unsafe and incorrect operation, but a type checker running at compile time only does not scan for divisions by zero in most languages, and then it is left as a runtime error. However, other kinds of static analyzes focus on proving value-based properties and not only type-based ones, and may be able to detect such errors.

```
let a = 6 in
let x = 0 in
let y = 3 in
if a mod 2 = 0 then 2 / x  else a * y + 1
```

Listing 2.2: A incorrect program accepted by type checking

```
(1) int i = 0;
(2) int j = 0;
(3) int err;
(4) while (5) (i < 100) {
   (6)err = rand();
   (7)i=i+1;
   (8)j=j+err;
   (9)
 }
(10) assert (j<=100);
(11)
```

Listing 2.3: a simple C program with a loop

We now present one such framework, which is widely used for static analysis purposes, which is Abstract Interpretation.

2.3.3 Abstract Interpretation

Abstract Interpretation is an automated framework of static program verification which was introduced in [CC76] and [CC77]. One of its many practical applications is to prove the absence of runtime errors during the execution of a program by considering an over-approximation of the program's behaviour and proving safety properties on this over-approximation.

Introductory Example

With the exception of the simplest examples, the verification of the safety properties is rarely complete. Automation of such an analysis requires the use of numerical methods to verify some approximation of properties. One major technique used in program verification is the over-approximation of the reachable sets of program points.

Consider the simple loop in Program 2.3: variables i and j start initialized at zero and the variable err is uninitialized. At each iteration, variable err is updated to a random value (in $[0; 1]$), i is incremented by one, and j is incremented by err. A forward invariant analysis would find that, at the end of the loop, $j \in [0; 100]$, $err \in [0; 1]$ and $i = 100$, hence the assertion holds. We explain in detail in the current section how to obtain such invariants.

2.3.4 Definitions

We now give the formal definitions for Abstract Interpretation which we illustrate with several examples.

Transition systems

Transition systems are used to describe the behavior of discrete systems. They are a very useful tool to formalize the semantics of programs.

Definition 16. *A transition system is a pair* (Σ, τ) *where* Σ *is a set of states, and* $\tau \subseteq P(\Sigma \times \Sigma)$ *a set of transitions between states.*

Notations 2.3.1

Given two states σ, σ', we note $\sigma \to \sigma' \in \tau$ the transition relation starting in the state σ, which after one execution step, leads to the state σ'.

Also we define the function $post : P(\Sigma) \to P(\Sigma)$ to compute the set of reachable states from a set of states $\mathcal{S}$ in one step:

$$post_\tau(\mathcal{S}) \triangleq \{\sigma' | \exists \sigma \in \mathcal{S} : \sigma \to \sigma'\}$$

Small-step operational semantics define methods to evaluate expressions one computation step at a time. They are generally used to describe the behaviour of a program in terms of the behavior of its parts (instructions, commands), providing a structural inductive view on its meaning. They can be seen as a transition system. To use transition systems to describe behaviours of programs, we often augment them with initial and final states:

Notations 2.3.2

Let $\mathcal{I} \subseteq \Sigma$ be a set of distinguished initial states and $\mathcal{F} \subseteq \Sigma$ be a set of distinguished final states. We also distinguish a set of blocking states $\mathcal{B}$ that have no successor: $\mathcal{B} \triangleq \{\sigma | \forall \sigma' \in \Sigma, \sigma \to \sigma' \notin \tau\}$ to model runtime errors, and correct program termination.

We can augment the blocking states, with two special states that we note α and ω, denoting respectively the correct and incorrect termination of a program. Our transition system can then be augmented by adding transitions from blocking states to either α or ω.

Many verification problems (absence of division by zero, unhandled exception etc.) can be reduced to inferring the reachable program states, and verifying some properties over those states. In order to do that, one needs to reason over not only one execution trace but the set of all possible ones, using a collecting semantics. An execution trace of a program is a representation of the execution of this program as a sequence of consecutive states.

Definition 17. *The collecting trace semantics is the semantics that computes the set of all possible traces. It can be defined according to a set of initial states* $\mathcal{I}$ *as follows:*

$$\mathcal{T}(\mathcal{I}) = \{s_0 \to \cdots s_n | s_0 \in \mathcal{I}, \forall i, s_i \to s_{i+1} \in \tau\}$$

From this set of all possible traces, we can now obtain the set $\mathcal{R}(\mathcal{I})$ *of all reachable states from* $\mathcal{I}$ *which is defined by:*

$$\mathcal{R}(\mathcal{I}) = \{s | s_0 \to \cdots \to s \in \mathcal{T}(\mathcal{I})\}$$

A proof of correction of a program (*i.e.* absence of runtime errors) then consists in proving that the intersection of the program's reachable states and the blocking states is empty:

Theorem 2. *A program starting in $\mathcal{I}$ is correct if and only if $\omega \notin \mathcal{R}(\mathcal{I})$*

Also note that the set of reachable states can also be expressed in fixpoint form, following [Cou78]:

$$\mathcal{R}(\mathcal{I}) = \text{lfp}(\mathcal{F}) \text{ with } \mathcal{F}(\mathcal{S}) \triangleq I \cup post_\tau(\mathcal{S})$$

To apply concretely this scheme to a program, we instantiate the states of our transition system to a control location in $\mathcal{L}$ and an environment in $\mathcal{E}$ mapping each variable to its value:

$$\Sigma \triangleq \mathcal{L} \times \mathcal{E}$$

If we go back to Program 2.3, we can now formulate its concrete invariant equation system. The goal is to associate to each control location $l \in \mathcal{L}$ a variable $\mathcal{X}_l$ with a value in $\mathcal{P}(\mathcal{E})$ which denotes the set of possible environments when the program reaches control state l. Note that this equation system can be derived automatically from the control-flow graph of the program.

$$\begin{aligned}
\mathcal{X}_1 &= \mathcal{I} \\
\mathcal{X}_2 &= \tau\{i = 0\}\mathcal{X}_1 \\
\mathcal{X}_3 &= \tau\{j = 0\}\mathcal{X}_2 \\
\mathcal{X}_4 &= \tau\{err = \top\}\mathcal{X}_3 \\
\mathcal{X}_5 &= \mathcal{X}_4 \cup \mathcal{X}_9 \\
\mathcal{X}_6 &= \tau\{i < 100\}\mathcal{X}_5 \\
\mathcal{X}_7 &= \tau\{err = rand()\}\mathcal{X}_6 \\
\mathcal{X}_8 &= \tau\{i = i + 1\}\mathcal{X}_7 \\
\mathcal{X}_9 &= \tau\{j = j + err\}\mathcal{X}_8 \\
\mathcal{X}_{10} &= \tau\{i \geq 100\}\mathcal{X}_5 \\
\mathcal{X}_{11} &= \tau\{j \leq 100\}\mathcal{X}_{10}
\end{aligned}$$

Concrete semantic is not computable: computing exactly the set of all possible behaviours of a program for all possible inputs is impossible if the size of the input is unbounded and, in practice, too hard when this size is big. Also even when the size of the input is limited, non-determinism makes the set of traces potentially unbounded. Even if we consider only one execution of a program, termination problems that can not be solved at compile-time arise: a major constraint that has to satisfy a static analysis is to terminate otherwise it is not usable in practice. Abstract Interpretation tackles this issue by computing an over-approximation of the behaviours of the program and also uses methods for enforcing termination of the analysis at the cost of a more approximate but yet sound program invariant.

Theorem 3. *Let p be a program starting in $\mathcal{I}$, $\mathcal{R}^\sharp(\mathcal{I})$ an over-approximation of the reachable states such that $\mathcal{R}(\mathcal{I}) \subseteq \mathcal{R}^\sharp(\mathcal{I})$, and $\mathcal{B}^\sharp$ an over-approximation of the erroneous states such that $\mathcal{B} \subseteq \mathcal{B}^\sharp$, then:*

$$\mathcal{R}^\sharp(\mathcal{I}) \cap \mathcal{B}^\sharp = \emptyset \rightarrow \mathcal{R}(\mathcal{I}) \cap \mathcal{B} = \emptyset \Leftrightarrow p \textit{ is correct}$$

Naturally, if over-approximations of some sets do not intersect, then the original sets do neither. The principal challenge of a static analyzer by Abstract Interpretation is to use an easy to compute approximation (*i.e.* easier than the concrete semantic), while trying to be precise enough to be able to prove: $\mathcal{R}^\sharp(\mathcal{I}) \cap \mathcal{B}^\sharp = \emptyset$.

To do that Abstract Interpretation relies on abstracting a concrete semantics into an abstract one and translating the operations of the program into the operation of an abstract domain's operation to make it computable. Depending on the abstract domain used, the abstract semantic will be more or less precise, and more or less expensive to compute.

2.3.5 Forward Analysis

Finding invariants at program points is made by solving a system of semantic equations, derived from the program to analyze and from the abstract domain used. The corresponding abstract elements used live usually in lattices, and are manipulated by the equation systems instead of the concrete values. Of course, analyzers can rely on widenings to speed up and guarantee the finding of an over-approximated invariant at the expense of accuracy sometimes, *i.e.* they reach a post-fixed point or a fixed point, but not necessarily the least fixed point of the semantic equations. In an analysis, each program instruction is associated with a transfer function that takes the element obtained from the instruction before, and returns this element modified according to the current element. The program is therefore associated with a composition of transfer functions, and computing its the reachability space corresponds to the computation of the smallest fixed point of this composition of functions. The calculation of the fixed point is necessary for analyzing loops or recursive function, or any instruction that induces a cycle in the control flow graph of the program. The functions associated with the instructions of the loop are applied multiple times, until the fixed point is reached, which is guaranteed by the use of the widening. Several iterative schemes are possible. For simplicity we apply every equation at each step, but other iteration scheme are possible (work-list, chaotic iterations[Bou93]).

2.3.6 Abstract Domains

We now present and formally define the principle of abstract domains which is a core notion in Abstract Interpretation.

As explained before, Abstract domains are based on the idea of abstraction. Logically, the abstraction relation can be seen as the inverse relation of the implication in the sense that if A implies B, then B abstracts A. For instance if we consider the predicate $P(x)$ which is true if the condition $x \equiv 0 \mod 4$ is verified, then the predicate $Q(x)$ which is true if the condition $x \equiv 0 \mod 2$ is verified, can be seen as an abstraction of the predicate P: $\forall x \in \mathbb{N}, P(x) \implies Q(x)$, and the resulting sets respect the inclusion $\{x|P(x)\} \subseteq \{x|Q(x)\}$

Using this idea, abstract domains exploit the notion of correspondence between sets of concrete values and abstract values through a Galois connection.

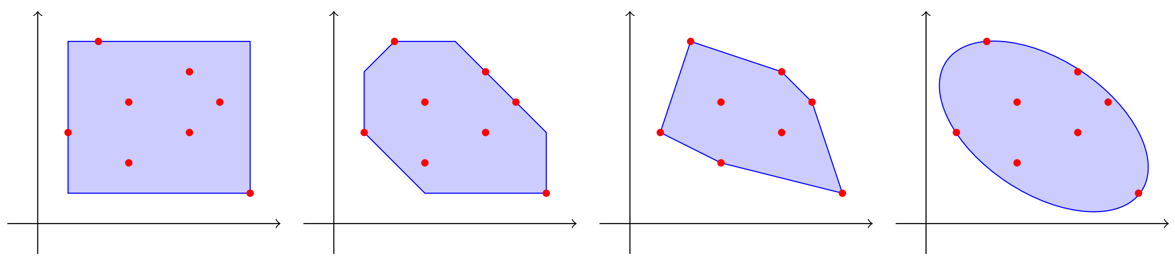


Figure 2.10: Abstraction of the same concrete element, with different abstract domains

Galois Connection

Abstract domains associate concrete properties (values, for example) to abstract properties (types, ranges of values). The abstract properties must always be sound approximations of the concrete ones. The concrete properties, and abstract values both live in lattices and when all concrete properties have a more precise corresponding abstraction, the correspondence is called a Galois connection. A Galois connection is a very interesting relationship between two partially ordered sets.

Definition 18. *Let $(A, \leq_A)$ and $(B, \leq_B)$ be two partially ordered sets. A monotone Galois connection between these posets is given by two monotone functions: $f : A \to B$ and $g : B \to A$ such that $\forall a \in A$ and $\forall b \in B$, we have:*

$$f(a) \leq_B b \equiv a \leq_A g(b).$$

Abstract domains embed a Galois connection, together with abstract versions of the operations of the program: Let D and $D^\sharp$ be respectively the domain of concrete values and the domain of abstract values, an abstract domain defines an *abstraction* function and a *concretization* function that form a Galois connection.

The *abstraction* function, noted α associates an abstract element to a concrete one. Figure2.10 shows an example of application of an abstraction function over the same set of points, with some of the most used abstract domains: the Boxes, Octagons, Polyhedra and Ellipsoids abstract domains.

$$\alpha_{D^\sharp} : D \to D^\sharp$$

Its symmetrical is the *concretization* function, noted γ. It associates to an abstract element a set of concrete values. It allows an analyzer to associate a concrete semantic for an abstract value. Figure2.11 shows an example of application of a concretization function from different abstract elements, with the previous abstract domains.

$$\gamma_{D^\sharp} : D^\sharp \to D$$

Note that Abstract domains may enjoy an abstraction function that forms a Galois connection together with the concretization, but this is not mandatory. In this case, every concrete element has an optimal abstraction. In the later definitions, we will not require an abstraction function, but only a concretization one.

An abstract interpreter is able to evaluate a program using abstract values instead of concrete ones. The result of the evaluation gives information about the behaviour of each step of the program. These information hold not only for a single execution but for all of the possible executions of the program in

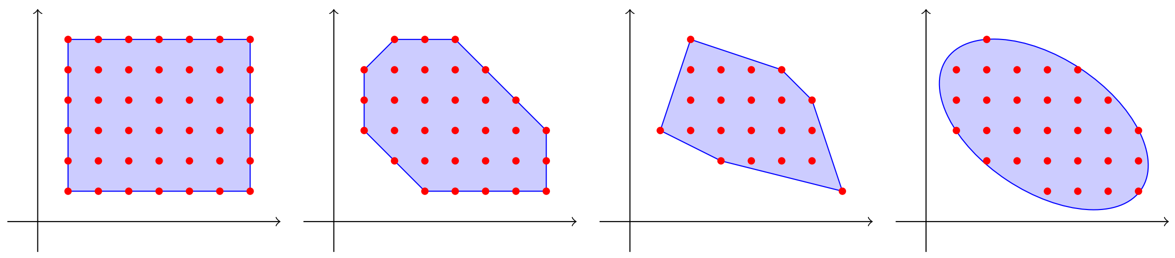


Figure 2.11: Concretization of different abstract elements

each step of the program. To be able to do so, the analyzer relies on an abstract domain, which instead of applying operations on concrete values, will rather define abstract operations. These abstract operations will reason on abstract elements and to be semantically correct, they must maintain the same behavior as their concrete counterpart in terms of property used for abstract values.

Abstract domains are thus defined according to a set of operations of a concrete semantic.

Definition 19. *Given a concrete semantic $\mathcal{L}$, abstract domains for this semantic are given by:*

- *a set $\mathcal{D}^\sharp$ of machine-representable abstract values,*
- *a partial order $(\mathcal{D}^\sharp, \sqsubseteq, \top^\sharp, \bot^\sharp)$ relating the amount of information given by abstract values,*
- *a concretization function $\gamma_{D^\sharp}$ giving a concrete meaning to each abstract element.*
- *an abstract operator $O^\sharp[r]$ for each rule r of the semantic $\mathcal{L}$*
- *abstract set operators $\bigcup^\sharp$ and $\bigcap^\sharp$, useful for computing the collecting semantic*
- *an algorithm to decide the ordering $\sqsubseteq^\sharp$*

Non-Relational Numerical Domains

A non-relational domain is an abstract domain that abstracts each variable separately, without any communication between these abstractions. They are unable to infer relationships between variables hence their name. A non-relational numerical abstract domain is given by: a subset of $P(\mathbb{V} \rightarrow \mathbb{I})$ (a set of environment sets) together with a machine encoding, effective and sound abstract operators, and an iteration strategy ensuring convergence of the analysis in finite time. It also must feature a concretization function $\gamma : D^\sharp \rightarrow P(\mathbb{I})$ giving a concrete meaning to each abstract element, which can be decomposed using a concretization function $\gamma' : (\mathbb{V} \rightarrow D^\sharp) \rightarrow P(\mathbb{V} \rightarrow \mathbb{I})$ at the variable level such that: $\gamma'(X) = \rho in \mathbb{V} \rightarrow \mathbb{I} | \forall v \in \mathbb{V}, \rho(v) in \gamma(X(v))$

An example of non-relational abstract domain is illustrated in Figure 2.12, where $\bigcap^\sharp$ and $\bigcup^\sharp$ are implicitly defined as the least upper bound and greatest lower bound.

From this lattice, we can easily define an abstraction function α as there is a Galois connection between the signs and the concrete values. Given a set of concrete values S, the abstraction function $\alpha : P(\mathbb{I}) \rightarrow D^\sharp$ is such that:

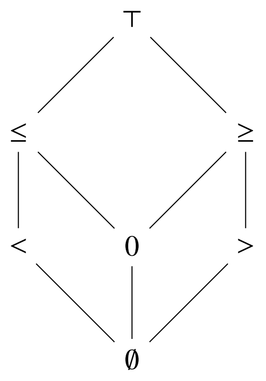


Figure 2.12: Lattice of the sign abstract domain

$$\alpha(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ < & \text{else if } \forall s \in S = s < 0 \\ > & \text{else if } \forall s \in S = s > 0 \\ 0 & \text{else if } S = \{0\} \\ \leq & \text{else if } \forall s \in S = s \leq 0 \\ \geq & \text{else if } \forall s \in S = s \geq 0 \\ \top & \text{otherwise} \end{cases}$$

2.3.7 State of the Art Tools

We end this section by quickly presenting some of the state of the art tools used in Abstract Interpretation:

- The static analyzer *Astrée : Analyseur statique de logiciels temps-réel embarqués* (real-time embedded software static analyzer) [BCC+02, BCC+03] aims at proving the absence of *Run Time Errors* (RTE) in programs written in the C programming language. On personal computers, such errors, commonly found in programs, usually result in unpleasant error messages and the termination of the application, and sometimes in a system crash. In embedded applications, such errors may have critical consequences.

- The static analyzer *AstréeA : Analyseur statique de logiciels temps-réel asynchrones embarqués* (real-time asynchronous embedded software static analyzer) [Min15] aims at proving the absence of *Run Time Errors* (RTE) in large scale asynchronous embedded software. AstréeA is built upon Astrée.

- More recently appeared MOPSA [MOJ18], a multi-language platform that simplifies the construction of semantic static analyzers defined by Abstract Interpretation. It provides a highly modular and extensible design: semantic abstractions of numeric values, pointers, objects, control flow, as well as syntax-driven iterators, are defined in small, reusable domains with loose coupling, that can be combined and reused to a greater extent than in state of the art.

- *Frama-C* is an extensible and collaborative platform dedicated to source-code analysis of C software. It has a plug-in system that can specialize different types of analysis. Among the existing plugins, *Evolved Value Analysis* (EVA) uses the abstract interpretation through a modular architecture including several abstract domains. *Frama-C* is used, for example, in the verification of critical code in the nuclear industry.

- Julia is a static analyzer for Java, Android and .NET. based on Abstract Interpretation methods. It analyses all possible execution paths of a program in a sound way. Moreover it is able to analyze bytecode and provides help for the correction of errors.

- The *Polyspace* static code analysis tool, for the C, C++, and *Ada* programming languages, uses formal methods to prove the absence of critical run-time errors under all possible control flows and data flows. They include checkers for coding rules, security vulnerabilities, code metrics, and several classes of bugs.

- The *Infer* analyzer of *Facebook* performs verification checks for resource leaks, reachability annotations, and concurrency race conditions. It provides support for Java, C, C++, and Objective-C , and is deployed at Facebook in the analysis of its Android and iOS apps.

2.4 Mixing Abstract Interpretation and Constraint Solving

The aim of this thesis is to build and exploit a tight collaboration between abstract interpretation and constraint programming. The main goal is to push the limits of these two domains by making them benefit from each other strength: Constraint programming provides powerful but computationally expensive algorithms to reduce domains with an arbitrary given precision whereas AI does not provide fine control over domain precision: in CP, we systematically refine a solution (split) without changing abstraction. While in AI the design of more expressive abstract domains is privileged. Incorporating some Abstract Interpretation mechanisms into a Constraint Programming paradigm would make constraint solvers benefit from the several abstract domains and their implementations that have been developed over the years in Abstract Interpretation, and we explain later on how it would allow the design of an efficient and generic constraint solver.

The first step of this work is to set the theoretical foundations of an hybrid method combining two substantially different paradigms. Once the interactions between CP and IA are well formalized, the next issue is to handle constraints of general forms and potentially non-linear abstract domains.

2.4.1 Common Points and Differences

In order to exploit the links between the two fields, let us identify their similarity and disparities.

About precision. Abstraction is key notion in Abstract Interpretation. It makes analyzes computable but less precise. In Constraint Programming, one never really sacrifices the precision, because the resolution continues until the desired accuracy is achieved. It is a big difference with Abstract Interpretation. In Constraint Programming, changing the abstraction just changes the way (and thus the efficiency with which) one reaches the goal. In Abstract Interpretation, changing the abstraction changes the result and whether the static analysis succeeds to prove the absence of error or not.

About computations. Numerical abstract domains of Abstract Interpretation are designed for machine integers and floatting points abstraction as the analyses are driven by a programming language's semantic. This is not the case in Constraint Programming as its main usage are driven by real world applications and thus, computations are made to approximate the real numbers.

About disjunctions. A usual way of representing disjunctions in Abstract Interpretation is the powerset abstract domain. Given an abstract domain, the powerset operator yields a new abstract domain which corresponds to the powerset of the original one and the operations are accordingly extended. This makes the new domain able to represent in the best possible way the concrete disjunctions. In Constraint Programming the exploration process is the natural of considering disjunctions. This is a more powerful, yet expensive, form of disjunction as it does not requite a concrete disjunction (a logical or $\vee$) to appear in the problem, but can be used as soon the representation loses accuracy.

About soundness and completeness. Soundness and completeness do not refer to same mathematical concepts in Constraint Programming and Abstract Interpretation. Generally speaking, being sound for a tool means that if the tool says that the desired property is true, then it is actually true. Symetrically, being complete means that everything that is true within the given theory is discovered by the tool. Soundness and Completeness are then interpreted differently depending on the type of property. In Abstract Interpretation, the desired property is generally *"the program has no incorrect behavior"*. To prove it, analyzers try to find invariants that respect the propery and that will hold for every possible execution trace of the program. Thus, an analysis that will lose execution traces is not sound anymore and considering an over-approximation of the execution traces is enough to prove the property. In Constraint Programming, the property of interest is *"this instance is a solution"*. As, the main goal is to find the solutions of a problem, being sound means that the results of the solving process are indeed solutions, and only solutions, of the constraint system. For a solver, considering an under-approximation of the states is sufficient to be sound. Basically, "sound" means the same thing in CP and AI, it is just the property of interest that changes. As this thesis focuses on the use of Abstract Interpretation techniques in Constraint Programming world, we keep the meanings that are generally used in Constraint Programming. This implies that a result that does not include all of the actual solutions of a problem is still considered as sound as long as is does not return non-solution instances. Mathematically, a sound result is an under-approximation of the solution space. In a similar way, completeness refers to the fact that all of the solutions are returned by the solving method, and adding non-solutions to the result does not break the completeness property. Mathematically, a complete result is an over-approximation of the solution space. In practice, solving methods are often complete and unsound, and static analyzers are often sound and incomplete.

In this thesis, we develop methods that can be, at the choice of the user, correct or complete.

2.4.2 Abstract Domains in Continuous Constraint Solving

A key point in the work we present in this thesis is the use of abstract domains in a Constraint Programming world. In Abstract Interpretation, abstract domains have been introduced to propose a computable approximation of program states [CC77]. For example, with the Interval abstract domain, each variable of a program is mapped to an interval with floating point bounds, and a program state is a Cartesian product of such intervals (box). An abstract domain is a partially ordered set (poset), where several operations can be made: transfer functions compute the result of an operation on an abstract element, the meet operator represent intersections of abstract elements, etc. Several recent research projects have been interested in the links between Constraint Programming and Abstract Interpretation. For instance, in [MBR16], the authors address the problem of proving an invariance property of a loop in a numeric program, by inferring automatically a stronger inductive invariant. The algorithm they present is based on both abstract interpretation and constraint solving. The computation of the effect of a loop is made using a numeric abstract domain and as in constraint satisfaction, it successively splits and tightens a collection of abstract elements until an inductive invariant is found. Another example, in an application framework closer to constraint programming, is [Tal18] where the author proposes a formal lattice-based definition of constraint programming, as well as a programming language to define exploration strategies for a constraint solver within posets. Also, abstract domains have already been extended to be used in a CP solver in [PMTB13b] and this work will serve as a starting point for us. We recall the main definitions and algorithms in this section. All of the definitions follow closely [Pel15].

Constraint Language

Let us consider a simple constraint description language, whose **BNF** grammar is presented in 2.13 and 2.14. The language uses a finite, fixed set $\mathcal{V}$ of real-valued and integer-valued variables, simple numeric expressions *arith-expr* and boolean expressions *bool-expr*, based on arithmetic comparison operators, or logical operators. We will be reasonning on the same language all along in this work.

Constraints are boolean expressions involving either comparisons of atomic arithmetical expressions, or conjunctions, disjunctons and negations of other constraints.

<bool-expr>	::=	<arith-expr> $\square$ <arith-expr>	$\square \in \{>, \geq, <, \leq, =, \neq\}$
	|	$\neg$ <bool-expr>	negation
	|	<bool-expr> $\vee$ <bool-expr>	disjunction
	|	<bool-expr> $\wedge$ <bool-expr>	conjunction

Figure 2.13: Boolean expression syntax

Arithmetical expressions include real constants, variables, usual operators over expressions and function calls among a list of predefined functions.

<arith-expr>	::=	c	$c \in \mathbb{R}$
	|	$\mathcal{V}$	variables
	|	<arith-expr> $\diamond$ <arith-expr>	$\diamond \in \{+, -, *, /, \%\}$
	|	- <arith-expr>	opposite
	|	*ident* '('<arith-expr>(',' <arith-expr>)*')'	function calls

Figure 2.14: Arithmetical expression syntax

We now define the semantic associated to this language using a big-step semantic. We instantiate the Abstract Interpretation framework on properties of numeric constraint satisfaction problems. The main intuition behind this idea is that contraint satisfaction problems can be seen as programs. Let us consider that a program state is given by an environment in $\mathcal{E}$ that maps each variable to a value: $\mathcal{E} \triangleq \mathcal{V} \rightarrow \mathbb{R}$ and let us suppose given an initial state $\mathcal{I}$ specified as a subset of environments $\mathcal{I} \subseteq \mathcal{E}$, a constraint satisfaction problem can then be seen as nested guards, and defining the corresponding semantic only requires defining the semantic function associated to guards $\tau\{\textit{bool-expr}\}$. The semantic function $\tau\{\textit{bool-expr}\}$, which is also called transfer function in Abstract Interpretation associates to a set of environments before an instruction the set of environments reachable after the instruction. In our case, it will only act as a filtering function. Boolean expression semantic is presented in Figure 2.15 and relies on semantics of expressions $[\![e]\!]\rho$ which is presented in Figure 2.16.

$$\begin{array}{lcl}
[\![b]\!]\rho & : & \mathcal{E} \rightarrow \mathcal{P}(\{t, f\}) \\
\hline
[\![e_1 \Box e_2]\!]\rho & \triangleq & \{t | \exists v_1 \in [\![e1]\!]\rho, v_2 \in [\![e2]\!]\rho : v_1 \Box v_2\} \cup \{f | \exists v_1 \in [\![e1]\!]\rho, v_2 \in [\![e2]\!]\rho : \neg(v_1 \Box v_2)\} \\
[\![\neg b]\!]\rho & \triangleq & \{t | f \in [\![b]\!]\rho\} \cup \{f | t \in [\![b]\!]\rho\} \\
[\![b_1 \vee b_2]\!]\rho & \triangleq & \{t | t \in [\![b_1]\!]\rho \cup [\![b_2]\!]\rho\} \cup \{f | f \in [\![b_1]\!]\rho \cap [\![b_2]\!]\rho\} \\
[\![b_1 \wedge b_2]\!]\rho & \triangleq & \{t | t \in [\![b_1]\!]\rho \cap [\![b_2]\!]\rho\} \cup \{f | f \in [\![b_1]\!]\rho \cup [\![b_2]\!]\rho\}
\end{array}$$

Figure 2.15: Boolean expression semantic

$$\begin{array}{lcl}
[\![e]\!]\rho & : & \mathcal{E} \rightarrow \mathcal{P}(\{\mathbb{R}\}) \\
\hline
[\![c]\!]\rho & \triangleq & \{x \in \mathbb{R}, x = c\} \\
[\![\mathcal{V}]\!]\rho & \triangleq & \{\rho(\mathcal{V})\} \\
[\![-e]\!]\rho & \triangleq & \{-v | v \in [\![e]\!]\rho\} \\
[\![e_1 \Box e_2]\!]\rho & \triangleq & \{v_1 \Box v_2 | v_1 \in [\![e_1]\!]\rho, v_2 \in [\![e_2]\!]\rho, v_2 \neq 0 \vee (\Box \neq / \wedge \Box \neq \%)\} \\
[\![name(x_1, ..., x_n)]\!]\rho & \triangleq & \{f(v_1, ..., v_n) | \forall i \in [1, n], v_i \in [\![e_i]\!]\rho, f = \text{runtime}(name)\}
\end{array}$$

Figure 2.16: Arithmetical expression semantic

Abstract Semantics

The concrete semantics is generally not computable as it manipulates values of $\mathbb{R}$. Even if it were not the case, for example if we were manipulating a more computer-friendly, finite data-type (such as machine integers or floating-point numbers), indeed the semantics becomes computable, but yet, it remains impractical because of its size. We tackle this problem in an Abstract Interpretation fashion by reasonning on abstract properties instead of concrete sets. We first define a set $D^\sharp$ of computer-representable properties, so-called abstract elements, together with a data-structure encoding and a partial order $\sqsubseteq^\sharp$ denoting the precision order between abstract elements, and a monotonic concretization function $\gamma : D^\sharp \rightarrow \mathcal{E}$.

To define an abstract domain over this language, one has to define abstract versions $F^\sharp : D^\sharp \rightarrow D^\sharp$ of all the operators defined in the concrete semantic, in our case, only $\tau^\sharp\{\textit{bool-expr}\}$.

Defining such an operator is enough to provide an abstract version of the big-step semantic, which is this time computable. Abstract semantics output an abstract invariant over-approximating the optimal invariant computed by the concrete semantics. Abstract domains should feature a concretization function (to associate a concrete meaning to an abstract elment) and may enjoy an abstraction function that forms a Galois connection. Some domains do enjoy it (such as intervals and octagons), but other, useful domains do not (such as polyhedra). For example there is no best polyhedron over-approximating a circle, as it would require such a polyhedron to have an infinite number of generators lying on the perimeter of the circle.

2.4.3 From Abstract Interpretation to Abstract Solving

Using the asbtract interpretation we have just defined, we are now going to augment it to produce a Constraint Solving framework. In order to do that we identify the main goals, and means to achieve those goals, and then we lift these to abstract domains. A classical solving method alternates two main steps: propagation and exploration. The abstract-solving method is defined by lifting up these operations to abstract domains. An abstract domain must thus feature a consistency operation, a split operator noted ($\oplus$) (which could also be called instantiation, for discrete domains). Also, in order to have a terminating process, we need to define a termination criterion, *i.e.* a precision predicate that we use to verify if an abstract element is too "small" to be refined. From these requirements, we augment Definition 19 to make it CP-compatible.

Definition 20. *Given a concrete semantic $\mathcal{L}$, abstract domains for this semantic are given by:*

- *a set $\mathcal{D}^\sharp$ of machine-representable abstract values,*
- *a partial order $(\mathcal{D}^\sharp, \sqsubseteq, \top^\sharp, \bot^\sharp)$ relating the amount of information given by abstract values,*
- *a concretization function $\gamma_{D^\sharp}$ giving a concrete meaning to each abstrac element.*
- *an abstract operator $O^\sharp[r]$ for each rule r of the semantic $\mathcal{L}$*
- *abstract set operators $\bigcup^\sharp$ and $\bigcap^\sharp$, useful for computing the collecting semantic*
- *an algorithm to decide the ordering $\sqsubseteq$*

Along with:

- *a size function τ [2],*
- *a splitting operator on $\mathcal{D}^\sharp$, $\oplus_{\mathcal{D}^\sharp} : \mathcal{D}^\sharp \to P(\mathcal{D}^\sharp)$,*
- *a consistency, along with an effective algorithm to reach consistency.*

Size Function

According to an ordered set E, a size function $\tau : E \to \mathbb{R}^+$ gives a metric on the size of an abstract element. It is used for the termination condition and should be designed such that an abstract element $e \in E$ is considered as a solution if $\tau_E(e)$ is less or equal than a parameter $r \in \mathbb{R}^+$. Usually r is very small and close to 0. Note that a size function may enjoy a monotonicity property: Let $E^\sharp$ an abstract domain and $(E, \leq)$ its underlying poset, then we say that $\tau_{E^\sharp}$ is monotonic if and only if:

$$\forall a, b \in E, a \leq_E b \implies \tau_{E^\sharp}(a) \leq \tau_{E^\sharp}(b)$$

Even though it is not mandatory, it is still a desirable feature of a size function.

[2] We have already used the symbol τ in the context of the definition of transition system to designate the set of transition relations, as it is customary to do so. Still, to stay in accordance with the notations used in [Pel15], we redefine it to designate the measure function of abstract elements

Split Operator

We call *split* the action of dividing an abstract element into smaller ones with respect to the order of the abstract elements.

Definition 21. *The splitting operator $\oplus_E$ associated with a poset $E, \leq_E$ must respect some conditions. Let $e \in E$, then:*

- $|\oplus_E(e)|$ *must be finite, ensuring the finite width of the search tree,*
- $\forall e_i \in \oplus_E(e), \tau_E(e_i) < \tau_E(e)$ *ensuring finite depth of the search tree (termination),*
- $\cup \oplus_E(e) = e$ *enforcing that splitting does not lose nor create solutions (completeness and soundness)*

Termination and Redundancy

Both the size function and the split heuristic must respect the decreasing scheme $\forall x \in \oplus_E(e), \tau_E(x) < \tau_E(e)$, as stated in Proposition 3.3.1 in [Pel12] so that there exists no infinite decreasing chain.

A supplementary condition which is not mandatory but gives better properties to the solving method is the *perfect cover* condition which guarantees that all the elements resulting from the split of an element e do not intersect with each other:

$$\forall a, b \in \oplus_E(e), a \neq b \implies a \cap b = \emptyset$$

This property ensures that no concrete instance is handled twice during the resolution.

Consistency Operator

The consistency is a property of satisfiability of an element according to a constraint or a set of constraints. Along with an effective algorithm to compute the consistent elements, it is used for the propagation phase of our solving method. We define it as follows:

Definition 22. *Given a set of variable $\mathcal{X}$ and their domains $\mathcal{D}$, let c be a constraint, and $D^\sharp$ an abstract domain and $L, \leq_L$ the underlying poset. An abstract element d is said to be $D^\sharp$-consistent with respect to c if and only if it is the least element of L containing all the solutions of that abstract element for c:*

$$\forall d' \in L, Sol(<\mathcal{X}, \mathcal{D}, \{c\}>) \subseteq \gamma(d') \implies d \leq_L d'$$

Abstract Domain Based Solving: the Algorithm

Using Definition 20 we can now define a solving method that does not depend on the abstract domain employed. This solving method follows the classical techniques from Constraint Programming that use propagation to reduce the domain of the variables and exploration to build smaller sub-problems from the original one.

Algorithm 3 Abstract solving

```
function SOLVE($\mathcal{D}, C, r$)                ▹ $\mathcal{D}$: domains, $C$: constraints, $r$: real
    cover ← $\emptyset$                                   ▹ solutions
    explore ← $\emptyset$                                 ▹ elements to explore
    $e$ =init($\mathcal{D}$)                                 ▹ initialization
    push $e$ in explore
    while explore $\neq \emptyset$ do
        $e$ ← pop(explore)
        $e$ ← filter($e, C$)
        if $e \neq \emptyset$ then
            if $\tau(e) \leq r$ then
                cover ← cover $\cup$ $e$
            else
                push $\oplus(e)$ in explore
```

By alternating propagation and exploration, Algorithm 3 builds a disjunction of abstract elements that covers the solution space. It uses two auxiliary functions: `init` $\in \mathcal{D} \rightarrow \mathcal{B}$, `filter` $\in \mathcal{B} \rightarrow \mathcal{B}$. Firstly, `init` creates an abstract element from the initial domains of the problem. Here we assume that the initial domain is exactly represented in the abstract domain. Then, `filter` corresponds to the propagation loop: it applies the propagator for each constraint in turn. Other orders are possible but we restrict ourselves to a simple round-robin for now.

This solving method works as follows: at each step, the current abstract element is tightened using the propagators on the constraints (function `filter`). After propagation, if the tightened abstract element is not empty, two cases are possible:

- if the abstract element is small enough with respect to a parameter r ($\tau(e) \leq r$), then it is added to the set of solutions `cover`
- if the size of the abstract element is larger than r and may contain solutions, then it is divided using a split operator $\oplus$ and the process is repeated on the resulting abstract elements.

Initialization

This step is the entry point of the solving method. It builds an abstract element form the possible values of the domains of the variables. In an Abstract Interpretation terminology, it is an abstraction function. In our particular case, there often exists a best abstraction function as the domains of the variables are expressed as a convex envelopp of some points, which our domains can express without losing precision.

```
function INIT($\mathcal{D}$)                      ▹ $\mathcal{D}$: domains of the variables
    $e'$ ← $\perp$
    for $i \in \mathcal{D}$ do
        $e'$ ← $e' \cup \alpha(i)$
    return $e'$
```

This procedure initializes by over-approximation the domains of the variables. This is the product of the possible abstractions of the values of the variables.

Filtering

```
function FILTER($e, C$)                    ▷ e : abstract element $C$: constraints
    $e' \leftarrow e$
    for $c_i \in C$ do
        $e' \leftarrow cons_A(e', c_i)$
    return $e'$
```

This procedure allows the filtering of an element according to a set of constraints. We filter the abstract element with each constraint in a *round-robin* order. Of course, different kinds of consistencies are possible, and this resolution method does not depend on the used consistency.

As is, Algorithm 3 builds a disjunction of elements that over-approximates the solution space. All of the elements are split until they are too small to be split again, according to the size function of the abstract domain employed. In [CJ09], Chabert & Jaulin showed a way to derive more expressive resolution techniques applying symbolic methods on the constraints. We follow this idea by adding a solving step to our resolution technique that stops the iteration as soon as an element satisfies entirely the constraint system, as shown by Algorithm 4

In order to do so, we define a third auxialiary procedure `satisfies` $\in \mathcal{B} \times C \rightarrow \{\text{true}, \text{false}\}$. This procedure checks whether an abstract element satisfies all the constraints, that is, if it contains only solutions. Of course, this procedure should have a sound behaviour, that is: in cas it use approximation, it is allowed to answer "false" in the case where the element satisfies all the same constraints, but never "true" in the case where the element violates at least one constraint. This function corresponds to a contractor as defined in [CJ09]. We incorporate to the algorithm a satisfaction test before going to the next iteration.

Algorithm 4 Abstract solving

```
1:  function SOLVE($\mathcal{D}, C, r$)          ▷ $\mathcal{D}$: domains, $C$: constraints, $r$: real
2:      sols ← $\emptyset$                                      ▷ sound solutions
3:      undet ← $\emptyset$                              ▷ indeterminate solutions
4:      explore ← $\emptyset$                               ▷ elements to explore
5:      $e$ =init($\mathcal{D}$)                                    ▷ initialization
6:      push $e$ in explore
7:      while explore $\neq \emptyset$ do
8:          $e \leftarrow$ pop(explore)
9:          $e \leftarrow$ filter($e, C$)
10:         if $e \neq \emptyset$ then
11:             if satisfies(e, $C$) then
12:                 sols ← sols $\cup\ e$
13:             else
14:                 if $\tau(e) \leq r$ then
15:                     undet ← undet $\cup\ e$
16:                 else
17:                     push $\oplus(e)$ in explore
```

This improved version of the algorithm works just like the first one except that after propagation, if the tightened abstract element is not empty, three cases are now possible:

- if the abstract element contains only solutions (function `satisfies`), then it is directly added to the set of solutions `sols`.
- if the abstract element is small enough it is added to the set of undeterminate solutions `undet` — i.e., the abstract elements which may contain both solutions and non-solutions, and are considered small enough to be left out of the search.
- Otherwise it is split ($\oplus$) and the process is repeated on the resulting abstract elements.

Not only the satisfaction test accelerates the resolution process as it avoids superfluous iterations but it also refines the results. At the end of the solving, we can now differentiate two kinds of elements, the ones that are undeterminate, which we stopped splitting because of the size termination criterion, and the ones that do satisfy the constraint system. Moreover, the implementation of this improvement is very simple as it does not require to manipulate the representation of the abstract elements, but can be defined in a generic way reusing the propagation.

Satisfaction Test

```
function SATISFIES($e, C$)                    ▷ e : abstract element $C$: constraints
    for $c_i \in C$ do
        if $cons_A(e, \neg c_i) \neq \bot$ then
            return false
    return true
```

This procedure takes an abstract element and a list of constraints and returns true if and only if the abstract element satisfies all the constraints. An element is considered to satisfy a constraint if it possesses an empty over-approximation with the negation of constraints. Here we could be more precise/efficient using an abstract domain specific satisfaction test, but this more generic method facilitates the implementation of new domains.

Termination of the Solving Method

To ensure the termination of the solver, we impose that any series of reductions, splits, and choices eventually outputs a small enough element for τ, following Definition 10 from [PMTB13b]:

The search procedure can be understood as a search tree browsing: each node corresponds to a search space and the children of a node are built by application of the split operator. Moreover, the set of nodes at a given depth corresponds to a disjunction approximating the solution. In addition, a serie of reductions and split correspond to a tree branch which, by the above definition is finite.

2.4.4 The AbSolute Solver

The AbSolute solver is an open source constraint solver based on abstract domains. It is built upon a rigorous theoritcal framework which is Abstract Interpretation, and features several techniques and classical heuristic from Constraint Programming. The solver is written in *OCaml* in a functional style and implements the solving method we have just presented. It is usable with several numeric abstract domains (Interval, Congruences, Octagon, Polyhedra) and domain combinators thus having several resolution means in one solver. It features a constraint problem description language, wery similar to the ones presented in Figure 2.15 and Figure 2.16, presented in Appendix section A.1, and a visualization tool that allows the user to picture the results of the solver, also described in AppendixA.3.

```
/* simple example with trigonometrical functions */
init{
  real x = [-10;10];
  real y = [-5;5];
}
constraints{
  y < (sin x) + 1;
  y > (cos x) - 1;
}
```

Listing 2.4: a simple example of CSP using AbSolute's description language

Listing 2.4 illustrates an instance of a constraint satisfaction problem described in AbSolute's syntax. Figure 2.17 shows the graphical output of its resolution using the box abstract domain.

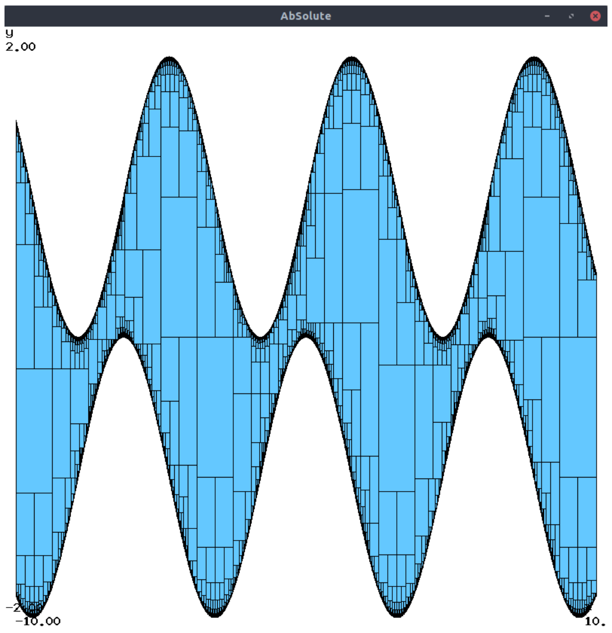


Figure 2.17: AbSolute's 2D graphical output of the solutions of Listing 2.4

AbSolute's standard ouput also indicates interesting properties about the results[3] such as the number of inner elements and outer elements.

Number of inner elements	6832
Total inner volume	40.823365
Number of outer elements	6528
Total outer volume	0.374564
Inner ratio	0.9909
Solving time	0.134s

[3]We omit here the complete list of solutions as there are a lot of them.

AbSolute relies on Apron[JM09], an abstract domains library dedicated to the static analysis of the numerical variables of a program by Abstract Interpretation. However, a lot of extensions to the libray were needed to make it fit Constraint Programming purposes (especially regarding the addition of split and size functions). The resolution is parametric in terms of precision, number of iterations, split heuristic etc, and can be done using one abstract domain, or several working together as we will see in Chapter 3. The solver can be used to solve problems containing real variables, integer variables or mixed integer-real problems as we will see in Chapter 4.

In this thesis, we will heavilly rely on this solver to experiment different techniques and implement the improvements we propose in this work. Each following chapter will be presented as the detail of a theoretical framework followed by a small implementation section and will be ended with an experimental section, in which we compare the results we obtain with different configurations of AbSolute or against other state of the art solvers.

Chapter 3

Abstract Domains and Domain Products for Constraint Programming

Abstract

A major issue in Constraint Programming is the fact that some solvers are very efficient on a specific constraint language, but can not (or hardly) be extended to tackle more general languages, both in terms of variable representation (discrete, or continuous) and constraint type (global, arithmetic ...). For instance, linear constraints are usually solved with linear programming techniques (LP). But it is then difficult to add new non-linear constraints to the problem: the user must either linearize or reformulate his/her constraint and keep the LP solver, switch to another solving method, or use an external solver in the LP method to refine the variable bounds with the non-linear constraints. In this chapter we adapt one very popular technique used in Abstract Interpretation to combine the power of several analysis, which is the Reduced product abstract domain. This domain is a good way to build both more precise analyses and extensible solver in a generic way. In this chapter, we present some of the abstract domains we use in our work, and their versions lifted to constraint solving, among which the polyhedra abstract domain. We also propose a modified version of the reduced product that is, we believe, better suited for Constraint Programming purposes. We will especially detail a particular instance of it which is the Box-Polyhedra reduced product. Finally we define general metrics for the quality of the results of a solver, and present a benchmark with respect to this metrics. Experiments show promising results and good performances.

Contents

3.1 Related Works

Solvers are in practice very different in the constraint languages they handle, yet, many different solving methods share common ingredients. In particular, they commonly use a consistency/propagation/bound computation algorithm, and a branching/splitting/instanciation process. Previous works [PTB14] showed how to unify these operations by re-defining a generic notion of domain, inspired from abstract domains of Abstract Interpretation. A generic solving process on CP abstract domains has been introduced in [PMTB13b], and implemented in the AbSolute abstract solver. Given an abstract domain (for instance, the continuous boxes), the generic solver calls the appropriate propagators and splits on this abstract domains (for instance, Hull consistency and splits for boxes). What is new, and important, in this abstract solver is that it is modular, that is, it can apply the same formal method, parametrized with different abstract domains in a generic way. The principal properties of the solver, which are termination, completeness and correctness, depend on the properties of the abstract domain it uses.

Several abstract domains have already been defined in Abstract Interpretation: Intervals [CC77], Polyhedra [CH78], Octagons [Min04, Min06a], Ellipsoids [Fer04]. Cartesian products of floating-point intervals already existed independently in CP to solve continuous problems (with Hull consistency[BGGP99]), and integer intervals for discrete constraints. In this case, the consistency level of the solver is often bound-consistency which is very relevant for interval based abstractions [vHYGD08, Pug98]. More recently, the Octagons have been defined for continuous constraints (with *ad-hoc* propagation and exploration heuristics) [PTB14].

These domains feature different precision (*e.g.* Octagons are more precise than Boxes, but less precise than polyhedra) and come at different costs (*e.g.* operation over Boxes generally have a linear complexity in terms of number of variables and operations over Polyhedra depend on the number of generators/constraint of the polyhedra which is unbounded). Also, some domains are designed to capture very specific properties over concrete values and ignore (or at least propose a very coarse approximation for) the other properties (*e.g.* Ellipsoids). Choosing which domain one has to use is not a trivial task as these facts must be taken into account.

Table 3.1 sums-up the charactercics of the some of the most widespread abstract domains: Boxes, Difference Bound Matrices [Dil90, ACD93], Ocatagons [Min01], Ocatahedron [CC04], Zonotopes [GP06,

	equation type	complexity	example	best α
Boxes	$\pm x_i \leq c$	$O(n)$	$x \leq 10$	✓
Difference Bound Matrices	$x - y \leq c$	$O(n^2)$	$x - y \leq 1$	✓
Octagon	$\pm x_i \pm x_j \leq c$	$O(n^2)$	$x + y \leq 1$	✓
Octahedron	$\pm x_i \ldots \pm x_k \leq c$	$O(2^n)$	$x + y - z \leq 3$	✓
Zonotopes	$x = \alpha_0^x + \sum_{i=1}^n \alpha_i^x \epsilon_i$	$O(c^n)$	$x = 3 + \epsilon_1 + 2\epsilon_2$	×
Polyhedra	$a \cdot x_i + \ldots + b \cdot x_j \leq c$	$O(c^n)$	$2x + 3y - 4z \leq 3$	×

Table 3.1: comparison of different abstract domains, regarding expressivity, complexity and featuring of a Galois connection

GP08, GGP09] and Polyhedra. They all are restriction of the polyhedra abstract domain, which may turn out to be too expensive in some cases. We present them with respect to the type of equations they are able to express, their space complexity in terms of variable, and the fact that they feature or not a Galois connection. The choice of the abstract domain to use is generally done before the analysis according to the target program and the analysis time granted. For example, a program supposed to be fast, like a compiler will probably not resort to a very expensive relational analysis but rather to an interval analysis or a weakly relational analysis using an abstract domain like Octagons.

Moreover, another kind of abstract domains exists, which is the abstract domains combinator. These abstract domains are built upon one or several abstract domains to improve and/or combine their precision. Such domains are very useful in practice as they easily enable the creation of very expressive combined domains in a modular and generic way. For example the Trace Partitioning abstract domain[RM07] allows the partitioning of execution traces of a program to be based on the history of the control flow (*e.g.* which branch of a conditional statement was taken), allowing a path sensitive analysis. It handles in a generic way the abstraction of the conrtol flow and uses another abstract domain (generally a numeric one) to handle the value analysis which will be more accurate thanks to the partitioning. Another very popular abstract domain combinator is the Reduced Product abstract domain[CCM11] which we rely on in this chapter.

A reduced product uses elements of the components of the product to represent conjunctions of properties from these domains. Additionally, operations in the reduced product apply a reduction operator to communicate information between the base domains, thus improving the precision. We present here a domain combinator that builds up an abstract domain from two base domains. Note that this domain composition generalizes well to an arbitrary number of abstract domains as the obtained abstract domain can be itself an element of a reduced product. This feature allows us a good composabilty: From an analyzer point of view, a product acts just like a regular domain. In order to be usable in a Constraint Programming framework, we will have to define a split operator and a precision function, along with a propagator. Also, a major difference between the reduced product we define here and the classical reduced product defined in Abstract Interpretation is that we introduce a hierarchy between the domains, one of them being specialized to a certain kind of problems only, and thus avoiding a redundancy of information between the two components of the product.

After we illustrate the shortcomings of a single abstract domain based resolution, we will recall the mandatory notions to understand how the reduced product operates, and will then focus on the use and design of a version of this product applied to constraint solving.

3.2 Introductory Example

The abstract domain parametrized resolution method allows using any abstract domain for solving problems. However, all domains are not equivalent, and each has its own strengths and weaknesses. Let us consider a concrete example: Example 3 gives a problem where both linear and non linear constraints are used.

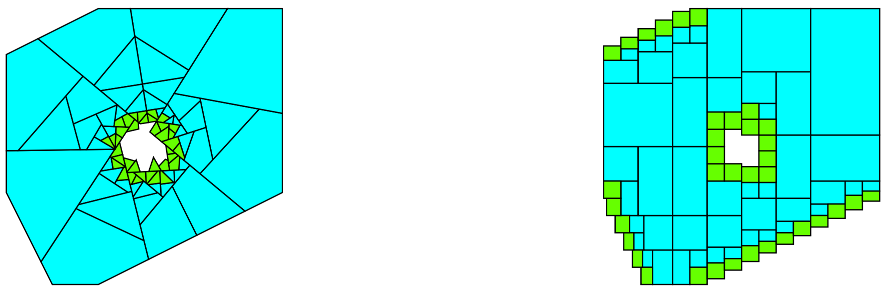
Example 3. *An example with 2 variables $(x, y) \in \mathbb{R}^2$ subject to 2 linear constraints and one non-linear constraint.*

- $x \in [-5; 5]$
- $y \in [-5; 5]$

Constrained with:

- $y \leq 2x + 10$
- $2y \geq x - 8$
- $x^2 + y^2 \geq 3$

Figure 3.1 illustrates the resolution of this problem using the Interval abstract domain, and the Polyhedra abstract domain.



(a) Solving using the Polyhedra abstract domain (b) Solving using the Interval abstract domain

Figure 3.1: Comparing the solving of a problem using different abstract domains

With the interval abstract domain, the time needed to reach the given precision is very small, 0.1 second, even though there is no constraint in the problem that intervals are able to express, that is constraint of the form $x_i \leq c$. The resolution with the Polyhedra abstract domain is much slower, 1.2 second, despite the fact that the first round of propagation discards the two linear constraints as they are encoded exactly inside a polyhedron. This important computation time is due to the fact that the polyhedra lose a lot of time to deal with nonlinear constraints because they require the use of an additional linearization step. We start from this observation to propose a combination of these domains, able to exploit the strengths of the domains, without having to suffer their weaknesses.

3.3 Reduced Product

Firstly, lets introduce Cartesian products. A Cartesian product is an operation that builds a set from two base sets, and the definition can be easily extended to any number of sets. For sets A and B, the Cartesian product $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. In terms of abstract domain, the Cartesian product is defined analogously, with its operations often being their counterparts in the base domain, applied component-wise.

Definition 23. *Let* $\{D_a, \sqsubseteq_a, \alpha_a, \gamma_a\}$ *and* $\{D_b, \sqsubseteq_b, \alpha_b, \gamma_b\}$ *be two abstract domains. The cartesian product* $\{D_{a\times b}, \sqsubseteq_{a\times b}\}$ *is an abstract domain ordered by the relation* $\sqsubseteq_{a\times b}$

$$
\begin{aligned}
(x_1, x_2) \sqsubseteq_{a\times b} (y_1, y_2) &\triangleq x_1 \sqsubseteq_a y_1 \land x_2 \sqsubseteq_b y_2 \\
\gamma_{a\times b}(x_1, x_2) &\triangleq \gamma_a(x_1) \cap \gamma_b(x_2) \\
\alpha_{a\times b}(v) &\triangleq (\alpha_a(v), \alpha_b(x_2))
\end{aligned}
$$

Where the abstraction function is naturally derived from the base domains and the concretization is the intersection of the concretizations of the base domains. For example, Figure 3.2 illustrates the application of the abstraction function for the Cartesian product of the Interval abstract domain with the Congruence abstract domain[Gra89].

Example 4. *Abstracting the the set of points* $s = \{(0, 1), (0, 2), (2, 2), (6, 2), (2, 3)\}$ *using a Cartesian product of Intervals and Congruences yields the following abstract value:* $([0, 6], 2\mathbb{Z}), ([1, 3], \mathbb{Z})$

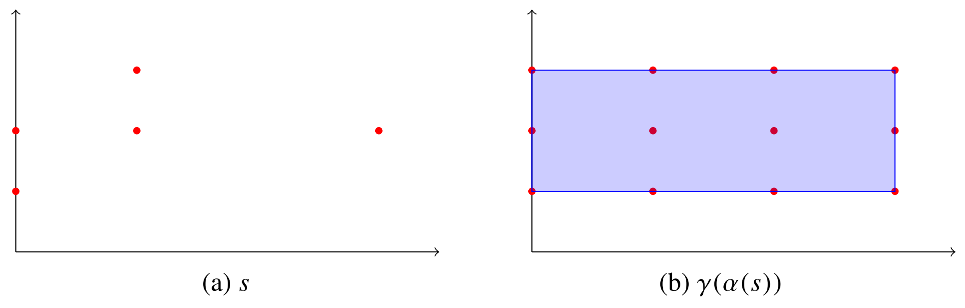

(a) s (b) $\gamma(\alpha(s))$

Figure 3.2: Cartesian product of interval and congruences

As is, a Cartesian product uses both the expressivity of its components independantly without having them communicating. Using a Cartesian product thus yields the same result as running two analyses with each abstract domain independently: a naive way of implementing abstract operators is component by component. This way, a loss of precision would arise from the control flow joins where information learned in a branch by one of the two domains is not communicated to the other as in the following example:

Example 5. *Consider the guard* $x < 10$ *where* x *is associated to the abstract element* $[0, 20]$ *with the intervals and* $4\mathbb{Z}$ *with the congruences, the transfer function filters effectively the interval, but can not refine the congruence. It gives us:* $[0, 10], 4\mathbb{Z}$*, inducing a loss of precision as* 9 *and* 10 *are values that* x *can not take, even though the fact that* $x \in [0; 8]$ *is expressible with boxes.*

Hence the use of a communication between components of the analysis. Also note that no loss of precision is due to the definition of the abstraction function α and the concretization one γ. A reduced product is a Cartesian product, which uses the same α and γ, but augmented with a reduction operator, which computes a better representation of an abstract pair. It is of particular interest as it allows propagating information between the two abstract domains, which refines both abstract elements at once.

Definition 24. *Given an abstract domain $\{D_a, \sqsubseteq_a, \alpha_a, \gamma_a\}$, A reduction operator $\rho_a \in D_a \to D_a$ is such that:*

$$\forall d \in D_a, \rho_a(d) \sqsubseteq_a d \wedge \gamma_a(\rho_a(d)) = \gamma_a(d)$$

This operator is used to propagate information between the two base components of the Reduced product. Its utility is to reduce the loss of precision induced by the abstraction, by filtering each of the component of an abstract value with the inconsistent values of the other components. In the presence of a Galois connection, one can define a better reduction operator in a genreic way as: $\rho = \alpha \circ \gamma$. If there is no Galois connection, a specialized reduction operator should be designed. For instance, if we once again consider the reduced product of the Interval abstract domain with the Congruence abstract domain, a reduction operator could reduce the following abstract values as shown:

Example 6.

$$\begin{aligned}
\rho([0,10], 2\mathbb{Z}) &= ([0,10], 2\mathbb{Z}) \\
\rho([0,10], 1+2\mathbb{Z}) &= ([1,9], 1+2\mathbb{Z}) \\
\rho([10,10], 2\mathbb{Z}) &= ([10,10], 0\mathbb{Z}+10) \\
\rho([0,10], 12+37\mathbb{Z}) &= \bot
\end{aligned}$$

Of course, according to Definition 24, the identity function is a correct reduction operator, yet most of the time very imprecise. In the following sections we present some abstract domains we use within a reduced product for which we are able to define a most precise reduction operator.

Notations 3.3.1

In the following, we note $D_{a \times b}$ the reduced product of abstract domains a and b

In the Constraint Programming terminology, a reduced product propagates information from one abstract element to another and vice-versa. It can be applied to any abstract domains, whether discrete or not, Cartesian or not. Applying reductions component-wise is more precise than using each abstract domain independently as it reduces the loss of precision of each abstract domain with the other's precision. With reduced products, we can solve a problem using different abstract domains, each applying its consistency independently and from time to time communicating their information to the other domains. Detailed examples are given in Section 3.6.

3.4 Lifting the Reduced Product to Constraint Programming

We now add to the definition of reduced product the requirement of our abstract solving method: a split operator, a precision function, and a consistency operator.

3.4.1 Product Splits

The splitting operation fits a products representation and can be naturally derived from the base domains.

Definition 25. *Given two abstract domains A and B, a reduced product split operator $\oplus_{A\times B} : A \times B \to \mathcal{P}(A \times B)$ is given by:*

$$\oplus_{a\times b}(A, B) = \{(x, y) | x \in \oplus_a(A), y \in \oplus_b(B)\}$$

Here, we simply map an abstract pair (a, b) of a product to all of the possible pairs resulting from the splits of a and b. This split is illustrated in the Example 7 with one integer variable, using a box-parity abstract domain (Where the letter 'O' stands for 'odd' and the letter 'E' stands for 'even')

Example 7 (Split operator with the Box-Parity reduced product)**.**

$$\begin{aligned}\oplus_{box,parity}([0, 10], \top) &= \{(x, y) | x \in \oplus_{box}([0, 10]), y \in \oplus_{parity}(\top)\}\\ &= \{([0, 5], O); ([0, 5], E); ([6, 10], O); ([6, 10], E)\}\end{aligned}$$

Example 7 shows that after a split, it is possible to perform another reduction operation to improve the precision. Indeed, after the split is performed, we obtain four elements that could be tightened using the reduction operator :

$$\{\rho([0, 5], O); \rho([0, 5], E); \rho([6, 10], O); \rho([6, 10], E)\} = \{([1, 5], O); ([0, 4], E); ([7, 9], O); ([6, 10], E)\}$$

Hence we propose to systematically follow the application of the split operator with the reduction operator, which gives us the following split operator:

Definition 26. *Given two abstract domains A and B, a self-reducing reduced product split operator $\oplus_{A\times B} : A \times B \to \mathcal{P}(A \times B)$ is given by:*

$$\oplus_{A\times B}(a, b) = \{\rho(x, y) | x \in \oplus_A(a), y \in \oplus_B(b)\}$$

Proposition 1 (Product split)**.** *The product split operator is a correct split operator according to Definition 21.*

Proof. Let us prove that $\oplus_{A\times B}$ is a finite, correct and complete split operator:

- We have $\forall a, b, | \oplus_{A\times B} (a, b)|$ is equal to $| \oplus_A (a)| * | \oplus_B (b)|$. Since both $\oplus_A$ and $\oplus_B$ respect Definition 21, $| \oplus_A (a)|$ and $| \oplus_B (b)|$ are finite and so is $| \oplus_A (a)| * | \oplus_B (b)|$.

- Correction and completion properties are trivially lifted to the produt as $\oplus_A$ and $\oplus_B$ are both correct and complete.

□

Of course other splits are possible as this one can be considered as over-aggressive as is would create potentially a lot of abstract elements. For example a round robin strategy performing a split on each element in turn can be possible, or a more chaotic strategy based on randomly choosing which abstract domain will perform the split is also possible. Also, if the two used domains have their abstract elements comparable, then we can simply choose the component to be split according the standard heuristic *largest-first*. In fact, even splitting with only one abstract domain, as explicited in Example 8, and neglecting the other is still a correct strategy, but would require to have an adapted precision function, that would take into account only the corresponding component of the product, to enforce the termination of our solving method.

Example 8 (Product split - Only left)**.**

$$\oplus_{A\times B}(a,b) = \{(x,y) | x \in \oplus_A(a), y = b\}$$

3.4.2 Product Precision Functions

The precision function also fits well a product representation and can be derived from the two base precision functions.

$$\tau_{a\times b}(A,B) = max(\tau_a(A), \tau_b(B))$$

Which ensures that the solving process continues as long as at least one of the abstract elements is big enough to be split. Also, depending on the product combination considered one could give a specialized size function that would be designed specifically for the combination of the underlying abstract domains.

3.4.3 Products Consistencies

The product's consistency is also naturally derived from the base domain's consistencies. Definition 22 can easily be extended to deal with products, as follows:

Definition 27. *Let c be a constraint, and $D^\sharp_{A\times B}$ a product of domains $D^\sharp_A$ and $D^\sharp_B$, and $(L_A, \leq_A)$, $(L_B, \leq_B)$ the underlying posets. An abstract element (a,b) is said to be $D^\sharp_{A\times B}$-consistent if and only if a is the least element of L_A containing all the solutions for c and b is the least element of L_b containing all the solutions for c:*

$$a = \alpha_A(c) \wedge b = \alpha_B(c)$$

3.5 Abstract Domains for Constraint Programming

In order to use the solving process presented previously, we need to define abstract domains. We present here in detail some of the most used ones in Abstract Interpretation, which we also implemented inside the AbSolute solver. Some are Cartesian ones (corresponding to representations existing in CP), relational ones (non-Cartesian) and even abstract domains corresponding to products of abstract domains.

In this chapter, we first give the Cartesian abstract domains corresponding to the representations existing in CP. We then describe some of the relational (non-Cartesian) abstract domains available in the Apron abstract domains library. We next define a variation of the classical reduced product abstract domain, which gets rid of the redundant information shared by the product's components. Finally we introduce two abstract domains corresponding to products of abstract domains, and measure their performances in the benchmark section.

3.5.1 Abstractions for Constraint Programming Consistencies

Constraint Programming techniques use several representations for the domains of the variables and exploit different methods for propagating constraints and splitting over these representations. These representations and the operations dedicated to them can be seen as abstract domains of Abstract Interpretation. The most widespread representation for the domains of the variables are integer sets and integer ranges for discrete problems, and real range with floating point bounds for continuous ones. We show in this section how these methods can be formalized as abstract domains as we use them in a unified framework. We detail here these representations and their operators as abstract domains.

Example 9 (Integer Cartesian Product)**.** *Let $\mathcal{V}$ be variables over finite discrete domains. We call integer Cartesian product any Cartesian product of integer sets in D. Integer Cartesian products form a finite lattice:* $\mathbb{S} = \{ \prod_i X_i \mid \forall i,\ X_i \subseteq D_i \}$

This kind of representation are very precise for the discrete and integer values as they allow the direct enumeration of possible values but become very expensive when the domains of the variables increase. Indeed, the size of these representations grow linearly with the size of the variables's domain. Hence, other representation may be needed in that case.

Example 10 (Integer Box)**.** *Let $\mathcal{V}$ be variables over finite discrete domains. We call integer box a Cartesian product of integer intervals in D. We note $[\![a,b]\!]$ with $a \leq b$ to represent the set $\{x \in \mathbb{Z} | a \leq x \leq b\}$. Integer boxes form a finite lattice:* $\mathbb{N} = \{ \prod_i [\![a_i, b_i]\!] \mid \forall i,\ [\![a_i, b_i]\!] \subseteq D_i,\ a_i \leq b_i \} \cup \{\emptyset\}$

This representation sacrifices precision in favor of efficiency and tractability of calculations. Here, the size of the representation does not depend on the size of the domains of the variables and is constant: only two values per variables are kept in memory, corresponding to its lower and upper bounds. However some operations may induce a loss of precision, *e.g.* the union of two disjoint boxes can not be represented exactly by one box and is the over-approximated with their bounding box.

Along with the machine representation and the partial order, according to Definition 20, we require a split operation for these representations to perform the exploration step of our solving process.

Example 11 (Split for the Integer Cartesian Product)**.** *Let $\mathcal{V}$ be variables over discrete domains, let v_i be the variable chosen for branching, and $x \in D_i$ its chosen value. The splitting operator is:*

$$\begin{aligned}\oplus_s(D_1 \times \cdots \times D_n) = \{\ & D_1 \times \cdots \times \{x\} \times \cdots \times D_n, \\ & D_1 \times \cdots \times D_i \setminus \{x\} \times \cdots \times D_n\}\end{aligned}$$

In the CP-terminology, this split is an instanciation of the variable v_i.

Example 12 (Split for the Integer Boxes)**.** *Let $(\mathcal{V})$ be variables over discrete domains, let v_i be the variable chosen for the split and $D_i = [\![a, b]\!]$ its domain. Let a be the chosen value for v_i. The splitting operator is:*

$$\begin{aligned}\oplus_{ib}(D_1 \times \cdots \times D_n) = \{\ & D_1 \times \cdots \times [\![a, a]\!] \times \cdots \times D_n, \\ & D_1 \times \cdots \times [\![(a+1), b]\!] \times \cdots \times D_n\}\end{aligned}$$

Example 13 (Split for the Boxes)**.** *Let $(\mathcal{V})$ be variables over continuous domains, let v_i be the variable chosen for the split and $D_i = [a, b]$ its domain as an interval. Let $h = \frac{a+b}{2}$ rounded to the nearest float. The splitting operator is:*

$$\begin{aligned}\oplus_b(D_1 \times \cdots \times D_n) = \{\ & D_1 \times \cdots \times [a, h] \times \cdots \times D_n, \\ & D_1 \times \cdots \times [h, b] \times \cdots \times D_n\}\end{aligned}$$

Integer Cartesian Products Abstract Domains. This abstract domain is based upon the lattice given in Example 9 and is ordered by inclusion. Its consistency corresponds to the generalized arc-consistency. The splitting operator instantiates a value to a given variable, and is given in Example 11. The precision function τ_s uses the size of the largest domain minus one:

$$\tau_s(D_1 \times \cdots \times D_n) = \max_i(|D_i| - 1)$$

If a solution is found, all variables are instantiated, all sets are singletons and τ_s is equal to 0. This ensures that for any $r > 0$ the solving process terminates.

Integer Boxes Abstract Domain. This abstract domain is based on the lattice in Example 10 with inclusion. It consists in a cartesian products of sets of consecutive integers. Its consistency is the bound-consistency. Its splitting operator assigns a variable to one of its bounds, and is given in Example 12. We use as precision function the length of the largest dimension, like with τ_s, if the element is a solution, τ_{ib} is equal to 0:

$$\tau_{ib}(\{a_1, b_1\} \times \ldots \times \{a_n, b_n\}) = \max_i(b_i - a_i)$$

3.5.2 A Relational Abstract Domain: the Polyhedra

Some abstract domains do not correspond to a Cartesian product. This means that they do not only keep one independant information per variable but they can also express relations about several variables. One such abstract domain is the polyhedra abstract domains which can be very useful in practice. We recall its definition in this section, then we augment it with our requirements to make it CP-compatible. We finish by measuring the performances of its implementation in AbSolute, within a reduced product.

The polyhedra domain [CH78] abstracts sets as convex, closed polyhedra. Modern implementations [JM09] generally follow the "double description approach" and maintain two dual representations for each polyhedron: a set of linear constraints and a set of generators. These are two complementary representations of the same data and the consistency between the two representations is maintained using Chernikova's algorithm [LV92]. A generator is either a vertex or a ray of the polyhedron. A ray corresponds to a vector along which, starting from any vertex of the polyhedron, any point is part of the polyhedron. However, the polyhedra we use do not have rays given that they are bounded.

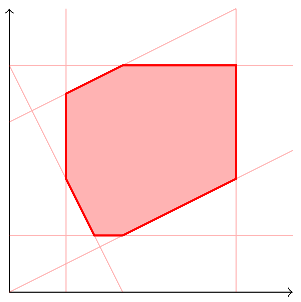
A polyhedron can then be defined either as :

- the convex hull of vertices and rays called generators. Given a finite set of points $\{v_1, \dots, v_k\}$ and their convex hull $\text{conv}\{v_1, \dots, v_k\}$, the polyhedron is then the Minkowski sum:

  $$P = \text{conv}\{v_1, \dots, v_k\} + \sum_{i=1}^{m} \alpha_i r_i$$

  with $\forall i, \alpha_i \in \mathbb{R}_+$

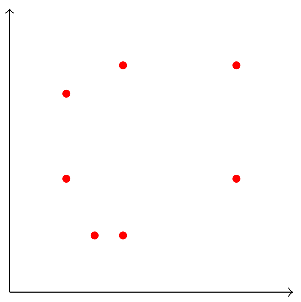
- or as the solution set of a finite number of half-space/hyperplane- constraints of the form: $A\vec{x}+b \geq 0$ or $A\vec{x} + b > 0$



(a) Linear constraint conjunctions defining a polyhedron

$$\begin{aligned}
&x \geq 1\\
&x \leq 4\\
&y \geq 1\\
&y \leq 4\\
&2 \times y - x \leq 6\\
&2 \times y - x \geq 0\\
&2 \times x + y \geq 4
\end{aligned}$$

(b) The corresponding equation system



(c) Generator representation of a polyhedron

Figure 3.3: Constraint and generator representation of a polyhedron

Figure 3.3 gives a visual of the different representations for a same polyhedron. The graphical representation 3.3a, the set of linear constraints 3.3b and the generators representation 3.3c.

Each representation has its benefits and its drawbacks, operators being generally easier to define in one representation than the other, and that is the reason why we keep the double representation in our case. We define the initialization, the consistency and the splitting operator of a polyhedron on the set of linear constraints. The size function is defined using the generators representation.

The abstract consistency in polyhedra is an important matter. It highly depends on the constraints nature. For linear constraints, the consistency can be directly computed by adding[1] the constraints to the polyhedron representation: this way, the algorithm which builds the polyhedron computes the generators in one shot, hence the consistent polyhedron for the considered constraints. In AbSolute, this step can be done once and for all during the initialization of the polyhedron.

For non-linear constraints, different approximations can be used, quasi-linearization [Min06b], linearization for the polynomial constraints [MFK+16], or computing the hull box, to name a few. As AbSolute is based upon Apron, it uses the same polyhedra consistency: the quasi-linearization.

The initialization of the polyhedron is performed using the set of linear constraints. The polyhedron can either be initialized using only the constraints corresponding to the domains in the CSP, or using also the linear constraints of the CSP.

Precision Function for Polyhedra

We can adapting to polyhedra the precision function we defined on boxes which gives us a definition based on the maximal Euclidian distance between pairs of vertices as shown on Figure 3.4.

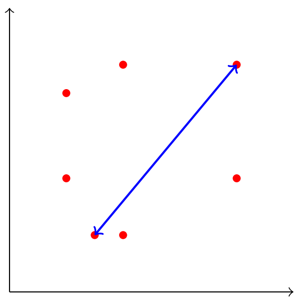


Figure 3.4: Maximal distance between two vertices

It is defined as follows, given a polyhedron P represented by its generators $\{g_0, g_1, ...g_n\}$:

$$\tau_p(P) = \max_{g_i, g_j \in P} ||g_i - g_j||$$

If we assume fixed the number of dimensions, computing $\tau_p(P)$ in a brute force way (checking all pairs of vertices of P) takes $O(n^2)$ time.

This problematic is well known as the *diameter problem* or the *furthest pair problem*. Given a set of point $P \in \mathbb{R}^d$ (with d the number of dimensions considered), it consists in finding the maximum distance δ between any two points of P (also called diameter of P). In the cases $d \in \{1; 2; 3\}$ there are several efficient algorithms[Ram97, Bes01] but they do not behave well when d grows larger.

As the precision function does not need to be accurate, we can use here an approximation algorithm for the diameter problem. For example, using the size of the largest dimension of the smallest enclosing box of the polyhedron can be sufficient.

[1]Of course, to keep a reasonable memory consumption, one has to remove the redundant constraints from the polyhedra (*e.g.* "$x < 5$" is redundant according to "$x < 2$" and does not need to be kept in the constraint representation)

Polyhedral Split

The split operator and the precision function should be designed in accordance so that the solving process terminates. The splitting operator duplicates the set of constraints representing the polyhedron, and then adds one constraint to each resulting set, the two added constraints being complementary, to ensure the fact that we do not lose solutions. Let $\mathcal{V}$ be variable over continuous domains, let P be a polyhedron, $\sum_{i\in\{1,n\}} \alpha_i v_i$ be the linear constraint corresponding to the split we want to perform and h the value where to split. The splitting operator is:

$$\oplus_p(P) = \{\ P \cup \{\textstyle\sum_{i\in\{1,n\}} \alpha_i v_i \leq h\}, P \cup \{\textstyle\sum_{i\in\{1,n\}} \alpha_i v_i > h\}\}$$

How is this constraint computed? Taking advantage of the fact that we have already computed the pair of most distant generators of the polyhedra with the precision function, we can reuse the computations for our splitting stragtegy: we first compute a line l that goes through these two generators, and from l, we compute an orthogonal line to it, l'. It is then sufficient to build the constraints corresponding to $l' > 0$ and $l' \leq 0$ which will be added into each of the two obtained polyhedra.

3.6 Specialized Reduced Product

In its classic form, the reduced product abstract domain handles a given statement with both of its components and then propagates the gathered information between them using the reduction operator. This is well adapted to a use in Abstract Interpretation but less to Constraint Programming: as there is no *a priori* knowledge on the statements to be interpreted during a program analysis, if a domain fails to find an invariant (or finds a very coarse one) on a given statement (*e.g.* because it uses expressions that are not well handled by the domain, for instance a quadratic expression abstracted with a linear conjunction domain), it may still be more accurate on the next statement. In a constraint satisfaction problem, all of the constraints are enunciated right from the begining of the problem. Then the solver works by performing several refinement iterations with the constraints to reduce the search-space using the abstract domains. But here, if a domain fails to filter the search-space using a given constraint because the constraint is not exactly representable with the abstract domain, then it will fail at every iteration of the solving process for the same reasons, inducing an important loss of time.

To tackle this issue, we define a specialized reduced product. We propose to establish a hierarchy between the components of the abstract domain: one of them will be a specialized component, *i.e.* it will only handle a certain kind of constraints, and the other one will be a default domain that will take care of the constraints the specialized domain does not handle.

Definition 28. *A specialized reduced product $D_{spe \times def}$ is a reduced product with a dispatch function $\delta : C \to \{true; false\}$ that given a constraint c assigns it to the specialized domain* `spe` *when the dispatch function returns true, or leaves it to the default domain* `def` *otherwise.*

In order to achieve this, we add to our definition of the reduced product a dispatch function that attributes a constraint to the specialized domain if the latter is able to encode it, and otherwise lets the default domain deal with the constraint. When an abstract element of the specialized domain satisfies a constraint, we then solve the remaining part of the problem using the default domain (if the default domain is itself a reduced product, we repeat the same process). Each time we propagate the constraints, then the reduced product with the specialized element is applied on the resulting element.

3.6.1 Specialized Consistency

The main idea of the specialized reduced product is to exploit the forces of the abstract domains without having to deal with their weaknesses. We thus define a specialized consistency following this idea, by attributing each constraint to one of the component of the product according the constraints the domains are able to handle efficiently.

Definition 29. *Let c be a constraint, and $D^\sharp_{spe\times def}$ a product of domains $D^\sharp_{spe}$ and $D^\sharp_{def}$, δ its dispatch function, and $(L_{spe}, \leq_{spe})$, $(L_{def}, \leq_{def})$ the underlying posets. An abstract element (a, b) is said to be $D^\sharp_{spe\times def}$-consistent if :*

$$\begin{cases} a = \alpha_{spe}(c) & \text{if } \delta(c) = true \\ b = \alpha_{def}(c) & \text{otherwise} \end{cases}$$

Also, note that thanks to the generic definition of the satisfaction test introduced in Chapter 2 and based on the reuse of propagation, there is no need to define a specialized satisfaction test for this reduced product.

3.6.2 Specialized Split

The split we have defined for products representation is a valid split for our specialized representation but can lead to unecessary splits when the constraints remaining to solve concern only one of the components of the products. Hence we propose here a split operator that takes into account a set of constraints to solve:

Definition 30. *Let C be a set of constraints, and $D^\sharp_{spe\times def}$ a product of domains $D^\sharp_{spe}$ and $D^\sharp_{def}$, δ its dispatch function, the split operator $\oplus_{spe\times def}$ is defined as follows:*

$$\oplus_{spe\times def}(a,b) = \begin{cases} \{(x,b) | x \in \oplus_{spe}(a)\} & \text{if } \forall c \in C, \delta(c) = true \\ \{(a,y) | y \in \oplus_{def}(b)\} & \text{if } \forall c \in C, \delta(c) = false \\ \{(x,y) | x \in \oplus_{spe}(a), y \in \oplus_{def}(b)\} & \text{otherwise} \end{cases}$$

Also, as the constraints can be removed from the constraint set as soon as they are proven to be satisfied, the components of the product handling those constraints are not split anymore from this point. This is due to the use of the dispatch function taking account the nature of the constraints, and if no constraint fit the specialized abstract domain, we simply stop splitting it. This is one of the main reasons we use a variant of the reduced product abstract domain instead of the regular one, as we want to avoid splitting elements systematically.

3.7 The Box-Polyhedra Instance

Now that we saw that an abstract domain can also correspond to a product of different abstract domains, we present here one such combination which is particularly powerful in AbSolute.

Remark *(Precision-cost trade-off)* When choosing an abstract domain to use when solving a problem, an interesting question to ask is if a polyhedron can represent exactly a hyper-cube or an octagon, why not use the Polyhedra abstract domain all the time during an analysis ? The answer is two-fold: firstly the Polyhedra abstract domain features in most of its operations a worst case complexity which is both exponential in space and time. In comparison, the interval abstract domain and the octagon abstract domain come respectively with linear and polynomial worst case complexity. Secondly, the polyhedra abstract domain does not feature a best abstraction function in the general case thus requiring some further development of linearization techniques to be able to use it in an actual implementation: as polyhedra can generally not compute a best approximation for non-linear spaces, one has to find a linear approximation of an expression over some range to be efficient. Thus, a solver tuned for Polyhedra only will be very slow in the general case (especially in presence of several variables and in presence of non-linear constraints), due to the expensive cost of the operation and to the additional processing step which involves linearization.

3.7.1 The Oracle

The Box-Polyhedra abstract domain is particularly useful when solving problems which involve both linear and non-linear constraints. Here, an interesting idea is to use the Polyhedra domain as a specialized domain working only on the linear subset of the problem, or the easy-to-linearize part of the problem. We use the Box domain as the default domain to solve the rest of the problem, *i.e.* expressions formed of non-algebraic functions like sin, log, etc.

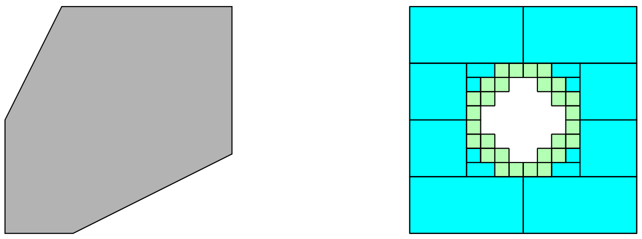
Linearization

Polyhedra must work with linear expression, hence the need to transform non-linear expression into linear ones in a sound way. We can use the intervalization [JM09] techniques based on replacing variables with their ranges. This approach yields an affine expression with interval as coefficients. As a heuristic, we decide to apply this technique for the polynomial expression that possess one unique variable with a degree superior to one, and handle the rest of the constraints using the default domain. This translates into the following oracle function:

$$\delta(c) = \exists x \in sup(c), deg(x) > 1, \wedge \neg \exists y \in sup(c), deg(y) > 1 \wedge y \neq x$$

Where $is\_polynomial(c)$ is an auxilliary procedure that computes true if the given expression is a multivariate polynomial. Also, $sup(c)$ is the set of variables of the constraint c and $deg(v)$ is the degree of the variable v.

Using this oracle, we can partition the memory representation of the problems accordingly. More precisely, let C_l bet the set of polynomial constraints, considered as linearizable, and V_l the set of variables appearing in C_l, and let C_{nl} be the set of non-linearizable constraints and V_{nl} the set of variables appearing in C_{nl}. We build an over-approximation of the space defined by C_l with the Polyhedra domain.


(a) Solving the linear part (b) Solving the non-linear part.

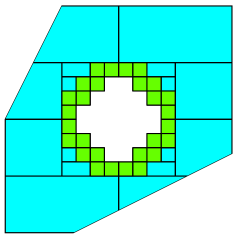

(c) Intersection of both resolutions

Figure 3.5: Example of the Reduced product of Box-Polyhedra.

By construction, this polyhedron is consistent with respect to C_l once it is created (conjunctions of linear constraints can be expressed with a convex polyhedron with no loss of precision). In short terms, the linear constraints are propagated once and for all at the initialization of the polyhedron. The variables V_{nl} appearing in at least one non-linear constraint are then represented with the box domain and the sub-problem containing only the constraints in C_{nl} is solved accordingly.

3.7.2 Consistency

Figure 3.5 gives an example of the Box-Polyhedra abstract domain applied on Problem 3 with both linear and non-linear constraints. Figure 3.5a illustrates the consistent polyhedron (for the linear constraints), Figure 3.5b the union of boxes solving the non-linear constraints, and Figure 3.5c the intersection of both abstract elements obtained with the reduced product.

As, by construction, the initial polyhedron is consistent for all the linear constraints of the problem, the operators in the reduced abstract domain box-polyhedra are defined only on the box part. Let $X = X_b \times X_p$ with X_b the box and X_p the polyhedron.

Definition 31 (Box-Polyhedra Consistency)**.** *Let $C = C_l \cup C_{nl}$ with C_{nl} (resp. C_l) the set of non-linear (resp. linear) constraints. The box-polyhedra consistent element is the product of the smallest consistent box including the solutions of C_{nl} with the initial polyhedron.*

This definition of the consistency is equivalent to the Product-consistency as, by construction, the initial polyhedron is consistent for the linear constraints (C_l).

Let $X = X_b \times X_p$ with X_b the box and X_p the polyhedron. The splitting operator splits on a variable in $V_{nl} = (v_1, \ldots, v_k)$ (in a dimension in X_b):

$$\oplus(X) = \oplus(X_b) \times X_p$$

Finally, the size function is:

$$\tau(X) = \tau(X_b)$$

Thus, we take advantage of both the precision of the polyhedra and the generic aspect of the boxes. Moreover, we bypass the disadvantages bound to the use of polyhedra.

Proposition 2 (Completness of solving with the Box-Polyhedra abstract domain)**.** *The solving method in Algorithm 3 with the abstract domain is complete.*

Proof. The Box-polyhedra consistency is complete; then, by Definition 10 in [PMTB13b], the abstract solving method using the Box-Polyhedra abstract domain is complete. □

3.8 Implementation and Benchmarks

We have implemented the reduced product presented in this chapeter inside the AbSolute solver and its performances have been compared with the `defaultsolver` of Ibex 2.3.1 [CJ09], on a computer equipped with an Intel Core i7-6820HQ CPU at 2.70GHz 16GB RAM running the GNU/Linux operating system.

3.8.1 Quality of a Cover

Before we present the results we obtained, let us discuss quality metrics. Of course, when solving continuous problems, different covers are possible and there does not exist a best one in the general case. Nonetheless, it is possible to compare the quality of two coverages produced by a solver according to several criteria among which:

Complete vs Correct. A solving method is generally either correct[2], *i.e.* its output contains only solution as illustrated by Figure 3.6a, or complete, *i.e.* its output contains all of the solutions, as in Figure 3.6b. When able to discriminate wheter an element satisfies a constraint system or not as shown in Figure 3.6c, we can produce a more interesting cover of the solution space that can be considered as both correct and complete depending on which set of elements one considers.

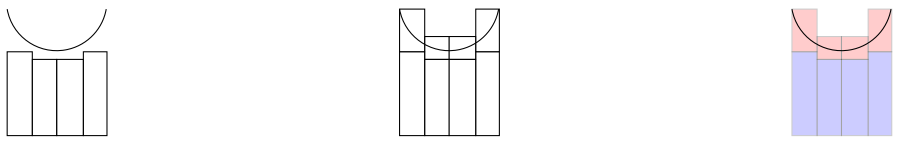

(a) Correct solving (b) Complete solving (c) Correct and Complete solving

Figure 3.6: Comparing a complete solving to correct solving

In that case, we call the elements that satisfy a given problem the inner elements. We call the ones that do not satisfy it (or can not be proven as satisfying element) the outer elements. Using this kind of property we can add a novel quality metric for the output of the solver which concerns the proportion of inner elements.

Definition 32. *Given a CSP P, and a set of abstract elements covering its solution space, we can distinguish the subsets I, of inner elements, and O of outer elements such that:*

[2]Here we use the term correct in the Constraint Programming sense

$$\forall x \in I, \gamma(x) \subseteq Sol(P)$$
$$\forall x \in Sol(P), \nexists i \in I, x \in \gamma(i) \rightarrow \exists o \in O x \in \gamma(o)$$

The first condition explicits that I under-approximates $Sol(p)$ and the second ensures that $I \cup O$ over-approximates $Sol(p)$.

Inner ratio. Exploiting the discrimination between inner elements and outer elements we can also assume that not all elements are equivalent and propose a volume metric for our abstract elements. On the entire volume of the cover, we compute the ratio from inner elements. When close to 1, this measure ensures a better reusability of the result: as the outer elements are not guaranteed to contain any solution, they may be subject to further refinement. Also certain applications could even discard those (*e.g.* when computing under-approximations of some space), and keep only the inner elements. Also, in the discrete case, the combinatorial enumeration can be limited to the outer elements, to be sure to have exactly all the solutions and only solutions.

(a) Inner ratio 0.2 (b) Inner ratio : 0.6

Figure 3.7: Comparing the inner ratio of two covers of the same solution space

Definition 33. *Given an abstract domain $D^\sharp$, let* vol : $D \rightarrow \mathbb{R}$ *a function that computes the volume of an abstract element, we want to maximize the ratio:*

$$\frac{\sum_{i \in I} \text{vol}(i)}{\sum_{i \in I} \text{vol}(i) + \sum_{o \in O} \text{vol}(o)}$$

This metric is the principal witness of the efficiency of the solver as it gives a concrete measure over the obtained cover: at the begining of the solving process, this ratio is equal to 0, as we begin the search within one *outer* element and as the solving goes, this ratio grows toward 1 as more and more inner solutions are found and the volume of outer elements is always decreasing. However, this metric requires the definition of a volume metric over our abstract element which is not straightforward. This metric is meant to be a good approximation of the number of concrete instances abstracted by a given element and sould be designed by taking into account the continuity of the search space.

Cardinality. The number of elements: the fewer elements will be in the cover space, the easier it will be to reuse the solver results. Figure 3.8 illustrates two covers of a same space that have different cardianalities. From a user point-of-view, the one with the smallest cardinality is easier to work with as it represents exactly the same amout of information in a denser way. Also, from a solver point-of-view, it shows that it has avoided superfluous exploration steps.

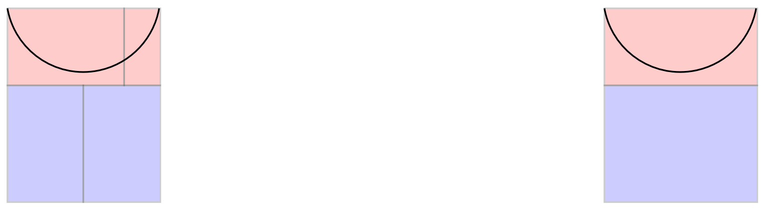

(a) 4 elements in the cover (b) 2 elements in the cover

Figure 3.8: Comparing cardinalities of two covers of the same solution space

Definition 34. *Given an abstract domain D, we want to minimize the number of abstract elements in the cover:*

$$|O| + |I|$$

Redundancy. If the results obtained do not intersect with each other, then we say of the resolution that it is non-redundant. This property guarantees that one does not treat several times the same concrete instantiations. Figure 3.9 gives an example of a redundant cover and a non-redundant one.

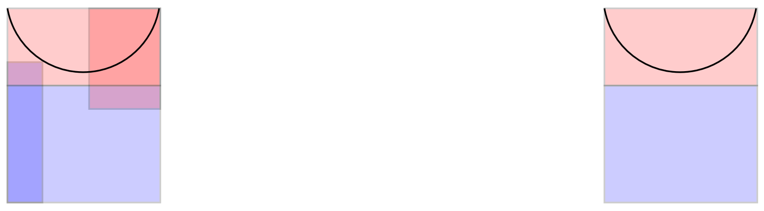

(a) Redundant cover (b) Irredundant cover

Figure 3.9: Redundant vs Irredundant

Definition 35. *Given an abstract domain equiped with a meet operation* $\cap$*, we want our cover C to satisfy the following property:*

$$\forall x, y \in C, x \cap y = \emptyset \vee x = y$$

We will use the metrics we have just defined to compare the results of both AbSolute and Ibex, and they will hold for the other chapters of this thesis. The resolution method that we define thereafter is as well adapted to a complete resolution than to a correct one. It is non-redundant and we define later on a technique of resolution able to reduce the number of elements returned by the solver, and maximize the inner ratio.

3.8.2 Experiments

Problems for this benchmark have been selected from the Coconut benchmark[3].

The selection has focused on problems with only linear constraints, only non-linear constraints or both as this is the main focus of this work. Also, we have fixed the precision (the maximum size of the solutions, with respect to the size metric of the employed domain) to 10^{-3} for all problems for both solvers, which is a reasonable precision for this kind of problems.

[3]Available at http://www.mat.univie.ac.at/~neum/glopt/coconut/

Table 3.2: Comparing Ibex and AbSolute with the interval domain.

problem	#var	#ctrs	time, AbS	time, Ibex	#sols AbS	#sols, Ibex
booth	2	2	**3.026**	26.36	**19183**	1143554
cubic	2	2	**0.007**	0.009	9	**3**
descartesfolium	2	2	0.011	**0.004**	3	**2**
parabola	2	2	0.008	**0.002**	**1**	**1**
precondk	2	2	0.009	**0.002**	**1**	**1**
exnewton	2	3	**0.158**	26.452	**14415**	1021152
supersim	2	3	0.7	**0.008**	**1**	**1**
zecevic	2	3	**16.137**	17.987	**4560**	688430
hs23	2	6	2.667	**2.608**	**27268**	74678
aljazzaf	3	2	**0.008**	0.02	**42**	43
bronstein	3	3	0.01	**0.004**	8	**4**
eqlin	3	3	0.07	**0.008**	**1**	**1**
kear12	3	3	0.043	**0.029**	**12**	**12**
powell	4	4	**0.007**	0.02	4	**1**
h72	4	0	**0.007**	0.012	**1**	**1**
vrahatis	9	9	0.084	**0.013**	**2**	**2**
dccircuit	9	11	0.118	**0.009**	**1**	**1**
i2	10	10	0.101	**0.010**	**1**	**1**
i5	10	10	0.099	**0.020**	**1**	**1**
combustion	10	10	**0.007**	0.012	**1**	**1**
st_miqp5	7	15	1.5	**174.661**	2135.269	**292**
hs5	2	1	0.05	**81.672192**	152.101994	**3501**
supersim	2	3	0.1	**7.116**	8.472	2

The first three columns of the table describe the problem: name, number of variables and number of constraints. The next columns indicate the time and number of solutions (*i.e.* abstract elements) obtained with AbSolute (col. 4 & 6) and Ibex (col. 5 & 7).

3.8.3 Analysis

We must define here the concept of solution for both solvers. Ibex and AbSolute try to entirely cover a space defined by a set of constraints with a set of elements. In Ibex, these elements are only boxes. In AbSolute, these are both polyhedra and boxes. Thus, the performance metric we adopt is, given a minimum size for the output elements, the number of elements required to cover the solution space. According to this metric, on most of these problems, AbSolute outperforms or at least competes with Ibex in terms of time and solution number. We justify the good results obtained by our method by two main facts: firstly, the linear constraints solving is almost immediate with our method. For example, the `booth` problem is composed of one linear constraint and one non-linear constraint. The linear constraint is directly representable with a polyhedron and thus, the solving process immediately finds the corresponding solution, while working only with boxes makes the search go through many splits before obtaining a set of boxes smaller than the required precision. Secondly, after each split operation, AbSolute checks for each resulting abstract element if it satisfies the set of constraints. If it is the case, the propagation/split phase stops for this element. This makes it possible to stop the computation as soon as possible. The `defaultsolver` of Ibex does not perform this verification and thereby goes much slower. This makes our implementation competitive on problems with only non-linear constraints. For the `exnewton` problem which only involves non-linear constraints (the resolution thus only uses boxes), we also obtain very good performances. Note that disabling the satisfaction verification in AbSolute leads to results with the same number of solutions as for Ibex, but still with a gain in time. For instance, with this configuration, on `exnewton` without the satisfaction check, we obtain 1130212 elements in 9.032 seconds.

Finally, regarding the solving time, the two methods have similar solving time. But we can notice that on bigger problems, using a polyhedron to represent the search space can be costly.

3.9 Conclusion

3.9.1 Contributions

In this chapter, we have shown that it is possible to define a constraint solving method based entirely on abstract domains, and introduced a well-defined way of solving problems with several domains together.

The framework we have presented is based on the idea of using an expressive domain able to encode exactly a certain kind of constraints, and a low-cost domain to abstract the constraints that can not be represented exactly (with no loss of precision) in the specialized domain. This allows us to get the best of both domains, while keeping the solver properties. The Box-Polyhedra product, for instance, is particularly adapted to problems with linear and non-linear constraints.

The principle is generic enough to add as many specialized domains as one wishes. For instance, the Congruence domain would allow us to efficiently represent constraints of the form: $a \equiv b \pmod{n}$. Therefore, one possible perspective of this work is to explore other specialized domains to broaden the set of constraints that can be efficiently solved with a dedicated method.

3.9.2 Perspectives

The promising results obtained with the AbSolute solver open the way to the development of hybrid solvers, at the frontier of Constraint Programming and Abstract Interpretation able to naturally handle different representations. In future work, we wish to improve our solving method by adapting and integrating advanced methods from the Constraint Programming literature, in particular specialized propagators for global constraints. The AbSolute solver is built on abstractions in a modular way, so that existing and new methods can be combined together. This opens many possibilities for mixing abstract domains, and new heuristics will be developed for choosing the appropriate abstract domain depending on the shape of the constraints. For example, the design of a product mixing standard abtsract domains of Abstract Interpretation and a more CP-oriented abstract domain which would implement a global constraint seems a promising idea. Ultimately, each problem could be automatically solved in the abstract domains which best fits it, as it is the case in Abstract Interpretation with a good distribution of constraints in relation to abstract domains, thus avoiding redundant computations and enjoying a better precision.

Chapter 4

Discrete and Continuous Abstractions for Constraint Solving

Abstract

Continuous problems and discrete problems are very different in nature and most constraint solvers are generally built to handle one kind of problem, either discrete or continuous but not both. While discrete solvers can heavily rely on exhaustive enumeration algorithms to reach to targeted consistency, for instance node consistency or arc consistency, this can not be achieved in the continuous case. Consequently, continuous solvers generally deploy other techniques, such as interval-based computations, to prune the search-space and they generally reach other consistencies such as hull-consistency or bound-consistency. In this chapter, we propose a framework that is able to handle discrete, continuous and mixed discrete and continuous problems in a unified way. To do this, we start from the HC4 algorithm, as it is appropriate for Cartesian representations, and we extend it to make it handle not only interval products, but discrete representations too. We introduce a generic representation for discrete values and continuous values, along with conversion rules inspired from implicit type conversion of numerical values as found in several dynamic programming languages. Finally, we extend our representation with some adapted exploration heuristics and we measure the efficiency of the obtained solving technique on the MINLPLib benchmark.

Contents

4.1 Related Works

Discrete constraint satisfaction problems are a lot more studied in literature than continuous ones. For example, the **XCSP3** competition's[LR19] format involves only integer variables, both for satisfaction and optimization problems. The distribution of discrete and continuous solvers illustrates this disparity as there exist several solvers that tackle discrete problems:

- Choco Solver[PFL14] is an Open Source Constraint Programming library written in Java. It is one of the most used discrete solvers.
- OR-Tools is an open source software suite, written in C++ mainly, for optimization, tuned for tackling problems such as vehicle routing, flows, integer and linear programming, and constraint programming.
- Concrete is a CSP constraint solver written in *Scala*. It solves CSP instances using depth-first search and arc-consistency (or weaker variants for propagation). It uses persistent data structures for managing domain states and some constraint states.

This list is not exhaustive and one can cite several other solvers/library such as Gecode[Sch02], GnuProlog [DAC12], Eclipse [AW07].... All of these solvers are very specialized and are able to handle a large variety of global constraints such as *allDifferent*, *allEqual*, *noOverlap* etc. Some solvers opt for a *portfolio* approach, like *Sunny-CP*[AGM15], which consists in combining a variety of different constraint solvers for solving a given problem. This enables a significant boost of the performance compared to single solvers, especially when multicore architectures are exploited, but these remain limited to discrete constraints only.

Yet, continuous constraint satisfaction problems are at the core of many real-world applications, including planning, scheduling, control, and diagnosis of physical systems such as car, planes, and factories. This raises the question of their scarcity in the Constraint Programming literature. These problems are generally called numerical CSPs. When dealing with a continuous problem, users are generally facing two options: either choosing to discretize its problem (potentially applying some scaling transformations to simulate a continuous behaviour), and then use a standard discrete solver to obtain the solutions. This option may require additional work with the results to retrieve an approximated continuous property. This can be unsatisfactory as the discretization is a workaround to avoid the use of a continuous resolution technique and has the disadvantage of being potentially incomplete.

The other option is to simply use a solver dedicated to continuous problems. Those are generally based upon a propagation/exploration technique to build a cover of the solution space, and use interval arithmetic. Among those are:

- Realpaver [GB06] is a nonlinear constraint solver. It implements interval-based computations in a branch-and-bound framework. Its key feature is to combine several methods from various fields: interval fixed-point operators, constraint propagation and local consistency techniques, local optimization using gradient descent methods and meta-heuristics, and several search strategies.

- Ibex [TC07] is a C++ numerical library based on interval arithmetic and Constraint Programming. Its goal is to find a characterization with boxes of a space implicitly defined by constraints.

One could also cite the less recent Declic Language [BGG97] and Global optimization language Numerica [vHMD97]. The main difference between those solvers and AbSolute is the fact that they are limited to interval based computations, which leads to covers consisting in box disjunctions while AbSolute builds disjunctions of abstract elements corresponding to an abstract domain given in parameter. Hence the cover can consist in Octagons, Polyhedra etc. Also, even if both kinds of applications, fully-discrete and fully-continuous can be handled with well established solvers, very few methods exist for the handling of mixed discrete and continuous problems, even though benchmarks for such applications exist. For example the Minlplib [BDM03] gathers several mixed problems. Minlp refers to optimization problems with continuous and discrete variables and nonlinear functions in the objective function and/or the constraints. Software developed for MINLP generally follow two approaches:

- Outer approximations/Generalized Bender's Decomposition. These algorithms alternate between solving a mixed-integer LP master problem and nonlinear programming sub-problems.

- Branch-and-Bound: Branch-and-bound methods for mixed-integer LP can be extended to MINLP with a number of heuristics added to improve their performance.

But these solutions remain workarounds as they are not fully dedicated to mixed discrete/continuous solving. Another example is [Ber10] where the author proposes a variant of the HC4 algorithm where integer variables are filtered according to an integrity constraint when encountered during the top-down part of the algorithm.

In this work, we propose a more general idea, that applies not only for integer variables but for any discrete representation. Firstly, we present it in the form of an interval arithmetic augmented with a flag that specifies if a given interval abstracts a continuous or a discrete space: this is in essence, a reduced product between the interval abstract and the very simple flag abstract domain which we illustrate as a lattice in Figure 4.1. We then refine this abstraction to further characterize the nature of the discrete space and improve precision using a revisited version of a well known abstraction of Abstract Interpretation which is the congruence abstract domain.

4.2 HC4 Propagation

Constraint in continuous problems are generally very distinct from constraints in discrete problems and analogously, propagators for continuous representations are very different from the ones for discrete

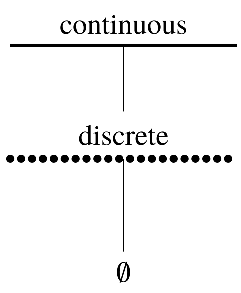


Figure 4.1: Lattice of the flag abstract domain

representations. As propagators strongly depend on the domains representations, defining a propagator for the *allDifferent* constraint would not make a lot of sense with continuous domains, as it would not filter effectively the domains of the variables (except in degenerate cases). We propose in this section to revise a well-known propagator of continuous representations, which is HC4, to make it handle discrete ones also.

Solving numerical systems manipulating real-values usually requires interval arithmetic based techniques due to the uncertainty implied by the rounding-errors. For example, Interval Newton algorithms generalize to intervals standard numerical analysis methods [JKDW01, MKC09]. The algorithm we are interested in is the **HC4** algorithm which given some domains and one constraint computes the smallest box, included in the domains of the variables, that encloses all of the solutions of the constraint.

Introductory Example

Let us consider a concrete example. Given variables x and y both with real-interval ranges in $[-2, 5]$ and a constraint c enforcing that $2 * y + x \leq 2$, we are going to compute the smallest box (a two-dimension square in this case) that contains all of the solutions for c. In order to do so, the first step is the interval evaluation of the expression, which is illustrated in Figure 4.2.

x	y
$[-2, 5]$	$[-2, 5]$

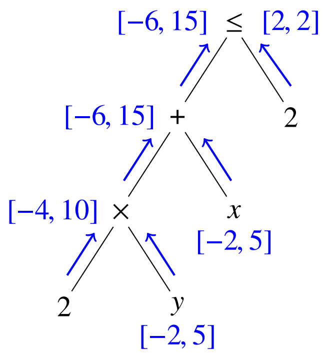


Figure 4.2: Interval evaluation of the expression $2 * y + x \leq 2$

Using a tree decomposition of the constraint, and a store containing the initial ranges for the variables, we evaluate both sides[1] of it using a bottom-up traversal of the tree. Each operation is mapped into its counterpart using interval arithmetic. With the given domains, the evaluation of the left side yields the

[1] In practice, one would more likely perform a symbolic transformation, removing the value 2 to both sides of the expression and evaluate only its left side. This simplifies the implementation work of the comparison operators, as every value will be compared to 0.

interval $[-6, 15]$, and the evaluation of the right side gives the singleton $[2, 2]$. Note that it is possible for this step to over-approximate the result. In this case, consistency is not reached but the completeness of the solving method still holds.

The second step of the algorithm is the top-down filtering part of an expression illustrated in Figure 4.3. Its goal is to refine the two intervals we obtained during the bottom-up step, by keeping only the values that may satisfy the condition. In our case, we want the smallest intervals $[i, j]$ such that:

$$\forall x \in i \cap [-6, 15], \exists y \in j \cap [2, 2], x \leq y \wedge \forall y \in j \cap [2, 2], \exists x \in i \cap [-6, 15], x \leq y$$

In other words, we keep all the values that have a chance to satisfy the constraints, and we remove all of the one that can not for sure satisfy it. Here, not all of the values in $[-6, 15]$ are less than or equal to 2, so we can remove the values from 3 to 15. We then propagate recursively the obtained intervals in both branches of the expression, and filter accordingly the annotated nodes: for each operation we had during the evaluation step, we can now refine its argument(s) now that we know an interval the result has to stay in. When a variable is reached, we update its domain bounds in the store if needed. Here too, if not all the values are removed from the intervals, the solving method is still complete even though consistency is not reached.

x	y
$[-2, 5]$	$[-2, \mathbf{2}]$

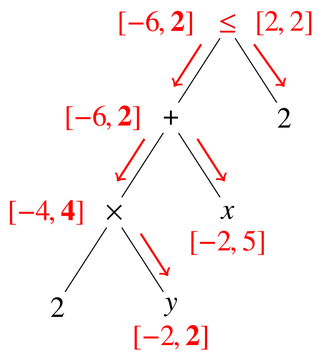


Figure 4.3: Filtering of the expression $2 * y + x \leq 2$

Applying these two steps only once leads generally to a larger hull than the smallest one enclosing all of the solution points. In particular in presence of several occurrences of the same variable, it would possibly lose precision. To reach Hull-Consistency, the bottom-up part and the top-down part have to be repeated alternatively until a fixpoint is reached[2].

4.2.1 A Non-Relational Abstract Domain for Constraint Solving

One interesting feature of this algorithm, which we exploit in this chapter, is that it holds not only for intervals but for any representation on which we are able to define the following operations: first, the usual arithmetic operations $(+, -, \times, /)$ used for the evaluation step. Second, the filtering tests $(<, \leq, >, \geq, =, \neq)$ that initiate the pruning of the expressions. Finally, the filtering versions of the arithmetic operations, used for propagating the pruning toward the leaf of the expression during the top-down step. Also,

[2]As our solving method does not need to reach Hull-Consistency at every iterations, and as the result is going to be subject to further splits and propagations, in the AbSolute solver we have made the choice to proceed to only one iteration of each step. This is assuming that the gain of speed will allow the resolution process to make more iterations, which will make up for the potential loss of precision

some set-theoretic operations are required: as the top-down step requires an intersection operation, the representation used has to form a meet-semi-lattice, *i.e.* for all elements x and y, the greatest lower bound of the set $\{x, y\}$ exists. Moreover, the representation is meant to be embedded into a non-relational abstract domain within a Cartesian product. Hence we need to define on it an order relation from which we can lift the order over the product representation: let n be the number of variables and x_i the element associated to the i-th variable of our Cartesian representation of the search space.

$$\prod_{i=1}^{n} x_i \subseteq \prod_{i=1}^{n} x'_i \leftrightarrow \forall i \in [1..n], x_i \subseteq x'_i$$

Size and Split for Non-Relational Abstract Domains

Altogether with the product representation, and the propagating function, the CP-version of our abstract domain requires a size function and an exploration heuristic. In accordance with the non-relational handling of constraints, we can define a variable selection heuristic and assign the splitting and measure work to the chosen representation. Such heuristic can be *largest-first* or *most-constrained* for example.

Definition 36. *Let us suppose that we have $\oplus_r$, a correct, complete, and irredundant splitting operator over the chosen representation and n the number of variables. Let v_i, $i \in [1.n]$ be the variable to split and d_i its associated representation: the splitting operations over abstract elements of our non-relational domains is then given by:*

$$\oplus(e) = \{d_1 \times \cdots \times x \times \cdots \times d_n | x \in \oplus_r(d_i)\}$$

Proposition 3. *Non-relational split. The non-relation split operator is correct, complete and irredundant.*

Proof. By definition of the Cartesian product, $\oplus$ is trivially complete, correct and irredundant. □

In a similar way, the size function computes the size of the domain of the chosen variable, which makes both the size function and the split heuristic consistent with the termination criterion.

Definition 37. *Let us suppose that we have τ_r, the splitting operator over the chosen representation and n the number of variables. The size function is then given by:*

$$\tau(e) = max\{\tau_d(d_1) \times \cdots \times \tau_r(d_n)\}$$

Defining other useful operations such as the set-theoretic ones ($\cup, \cap$) is easy as they are naturally lifted to a product representation using pairwise operations over the variables.

We are now going to explicit one such representation, which is well suited for continuous ranges of values.

4.3 Interval Representation

A natural abstraction of contiguous sets of values is intervals. They allow the representation of the values they contain using only a lower bound and an upper bound.

4.3.1 Continuous Arithmetic Challenges

Machine-representable types such as integers and floating point numbers are finite and discrete. Since real numbers $\mathbb{R}$ are infinite and continuous, they cannot be represented in computer architectures, in the general case. This makes the problem of defining an abstraction for continuous values not trivial and in particular we have to address two problems: fixed precision and rounding errors.

Fixed precision fixes the number of digits/bits used to represent a certain number. This limits the range of all the operations to certain values (generally $\pm 2^{32}$ or $\pm 2^{64}$) which causes the problem of arithmetic overflow. An arithmetic overflow error occurs when an arithmetic operation attempts to create a numeric value that is outside a given range. A solution to postpone (and in practice almost completely eliminate) the overflowing problem is the use of arbitrary precision representations. Those grow along with the number to represent and the calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system. Of course, their unbounded size makes them slower than fixed-size representations and they are principally used in applications where the speed of arithmetic is not a limiting factor, or where precise results with very large numbers are required.

Rounding errors occur when the result of a calculation is not exactly representable with the type being used. For example the result of $\frac{1}{3}$ would require an infinite number of decimals to be represented and will be rounded toward a close representable value depending on the rounding-mode of the processor being used. When a sequence of calculations involving round-off errors are made, errors may accumulate, sometimes completely obfuscating the calculation. A solution to limit rounding errors is the use of more symbolic representations such as rational numbers which are numbers that can be expressed as a ratio of two integers. This avoids rounding errors due to certain arithmetic operations such as division.

4.3.2 Continuous Intervals

To represent real ranges of values, we can use pairs of bounds which correspond to included or excluded endpoints for an interval. The bounds should belong to a subset of $\mathbb{R}$ to be comparable with reals, and also have to be representable exactly on computers.

Definition 38. *Let $\mathbb{B}$ be a subset of $\mathbb{R}$. A real interval with bounds in $\mathbb{B}$ is a pair $((l, b_l), (u, b_u))$ where $(l, u) \in \mathbb{B}^2$ are the endpoints of the interval and $(b_l, b_u) \in \{true, false\}^2$ are flags denoting the inclusion (true) or exclusion (false) of the endpoints of the interval. The concretization γ of such an interval is given by:*

$$\begin{aligned}
\gamma((l, true), (u, true)) &= \{x \in \mathbb{R} \mid l \leq x \leq u\} \\
\gamma((l, true), (u, false)) &= \{x \in \mathbb{R} \mid l \leq x < u\} \\
\gamma((l, false), (u, true)) &= \{x \in \mathbb{R} \mid l < x \leq u\} \\
\gamma((l, false), (u, false)) &= \{x \in \mathbb{R} \mid l < x < u\}
\end{aligned}$$

Notations 4.3.1

- We note $B^2_{\mathbb{R}}$ the set of real intervals with excluded or included bounds in B, where $\mathbb{B} \subseteq \mathbb{R}$.
- We note $B^2_{\mathbb{Z}}$ the set of integer intervals with bounds in $\mathbb{B}$, where $\mathbb{B} \subseteq \mathbb{N}$.

Note that the interval arithmetic we present here does not depend on the computer representation used for the bounds. Those only have to satisfy some requirements, such as implementing a total order, providing arithmetic operation etc. Moreover, due to the possibility of rounding-error when using some operations, we require for each operator $\square$ two version of it: one that rounds the result toward ∞ when the exact result can not be computed, which is noted $\square^+$ and one that rounds the result toward $-\infty$ which is noted $\square^-$[3]. The image of an interval under a function f should be computed using the appropriate version of the operators, depending on the monotonicity of f. For example, if f is a monotonically increasing function, then the image of and interval $((l, b_l), (u, b_u))$ under f is given by $((f^-(l), b_l), (f^+(u), b_u))$.

In the Absolute solver, several types of bound are implemented, with careful handling of rounding errors including integer, floating-point and multi-precision rational bounds.

We have to define a meet relation, noted $\cap$ over our representation. Here, when dealing with operands with only closed bounds, the result has closed bounds too. When at least one of the operand has one of its bounds open, the result will have the corresponding bound open too, as the $\cap$ operator must respect the relation:

$$\forall x \in a \cap b, x \in a \wedge x \in b$$

When one of the operand has both its bounds open, the result will have both its bound open too for the same reasons. Hence, using Booleans to encode the kind of bounds (open or closed) allow us to use the conjunction operator $\wedge$ to describe the behaviour of the bounds.

Definition 39. *A meet relation* $\cap : B^2_{\mathbb{R}} \times B^2_{\mathbb{R}} \to B^2_{\mathbb{R}} \cup \emptyset$ *is a binary operation such that* $((l_1, b_{l_1}), (u_1, b_{u_1})) \cap ((l_2, b_{l_2}), (u_2, b_{u_2})) = r$ *where :*

$$r = \emptyset \text{ if and only if } l_1 > u_1 \vee l_1 = u_1 \wedge \neg(b_{l_1} \wedge b_{u_1}) \vee l_2 > u_2 \vee l_2 = u_2 \wedge \neg(b_{l_2} \wedge b_{u_2})$$

otherwise,

$$r = ((l_3, b_{l_3}), (u_3, b_{u_3})), \text{ where}$$

$$l_3 = max(l_1, l_2) \qquad\qquad u_3 = min(u_1, u_2)$$

$$b_{l_3} = \begin{cases} b_{l_1}, & \text{if } l_1 > l_2 \\ b_{l_2}, & \text{if } l_1 < l_2 \\ b_{l_1} \wedge b_{l_2} & \text{if } l_1 = l_2 \end{cases} \qquad\qquad b_{u_3} = \begin{cases} b_{u_1}, & \text{if } u_1 < u_2 \\ b_{u_2}, & \text{if } u_1 > u_2 \\ b_{u_1} \wedge b_{u_2} & \text{if } l_1 = l_2 \end{cases}$$

We can also define a join relation, which is not mandatory, but will prove handy to define some arithmetical operators. It corresponds to the convex hull of its two argument and may thus over-approximate the result when applied on disjoint intervals. It has to respect the relation below:

$$\forall x \in a \cup b, x \in a \vee x \in b$$

Symmetrically, we can use the disjunction operator $\vee$ to describe the behaviour of the bounds.

[3]Most processors today allow to set the rounding mode used.

Definition 40. *A join relation* $\cup : B_{\mathbb{R}}^2 \times B_{\mathbb{R}}^2 \rightarrow B_{\mathbb{R}}^2$ *is a binary operation where* $((l_1, b_{l_1}), (u_1, b_{u_1})) \cup ((l_2, b_{l_2}), (u_2, b_{u_2})) = ((l_3, b_{l_3}), (u_3, b_{u_3}))$ *is such that:*

$$l_3 = min(l_1, l_2) \qquad u_3 = max(u_1, u_2)$$

$$b_{l_3} = \begin{cases} b_{l_1}, & if\, l_1 < l_2 \\ b_{l_2}, & if\, l_1 > l_2 \\ b_{l_1} \vee b_{l_2} & if\, l_1 = l_2 \end{cases} \qquad b_{u_3} = \begin{cases} b_{u_1}, & if\, u_1 > u_2 \\ b_{u_2}, & if\, u_1 < u_2 \\ b_{u_1} \vee b_{u_2} & if\, l_1 = l_2 \end{cases}$$

4.3.3 Interval Arithmetic and Filtering

Now that we have defined the set-theoretic operators over our representation, lets define an arithmetic over it. The algorithm we use can be decomposed into two steps: one evaluation step and one filtering step. We now detail the evaluation operations and filtering operation for real intervals.

We extend the interval arithmetic presented in the preliminary Section 2.1.3 tho handle included and excluded bounds.

Interval Arithmetic Evaluation Rules

The evaluation step of the algorithm computes an interval which contains all the possible values for the expression evaluated. It requires to define arithmetic operations over the used representation, and, in order to be sound, these operations have to encompass the behaviour of their concrete counterparts. For instance, for every binary operator $a \Box b$, we have to define an evaluation rule noted $\Box_{eval}$. This rule has to compute an over-approximation of the result of $a \Box b$ in order to preserve the soundness property.

$$\Box_{eval}(a, b) = i \rightarrow \forall x \in a, \forall y \in b, x \Box y \in i$$

Once again, we can note that an evaluation rules that returns the interval $[\infty, -\infty]$ is valid for all operator, but still, in order to be precise, we try to compute when it exists the smallest over-approximation. An evaluation rule $\Box_{eval}$ is optimal if it respects the following property:

$$\Box_{eval}(a, b) = min\{i | \forall x \in a \forall y \in b, x \Box y \in i\}$$

Provided that we have arithmetic operation over bounds of our continuous intervals, that can be rounded toward ∞ and $-\infty$, we can now define arithmetic operation, over our continuous intervals with a sound handling of round-off errors. When the result for a bound can not be exactly computed, it should be rounded toward ∞ (respectively $-\infty$) if it concerns an upper (respectively lower) bound. We have implemented in the AbSolute solver all the evaluation rules for the operations presented in Appendix A.1. For sake of concision we do not give all the evaluation rules in this section but illustrate the method on some of them:

- $-_{eval}((l, b_l), (u, b_u)) = ((-^{-}u, b_u), (-^{+}l, b_l))$
- $((l_1, b_{l_1}), (u_1, b_{u_1})) +_{eval} ((l_2, b_{l_2}), (u_2, b_{u_2})) = ((l_1 +^{-} l_2, b_{l_1} \wedge b_{l_2}), (u_1 +^{+} u_2, b_{u_1} \wedge b_{u_2})$

- $((l_1, b_{l_1}), (u_1, b_{u_1})) *_{eval} ((l_2, b_{l_2}), (u_2, b_{u_2})) = (min(ac, ad, bc, bd), max(ac, ad, bc, bd))$ where,

$$\begin{aligned} ac &= (l_1 *^- l_2, b_{l_1} \wedge b_{l_2}) \\ ad &= (l_1 *^- u_2, b_{l_1} \wedge b_{u_2}) \\ bc &= (u_1 *^- l_2, b_{u_1} \wedge b_{l_2}) \\ bd &= (u_1 *^- u_2, b_{u_1} \wedge b_{u_2}) \end{aligned}$$

- $exp_{eval}(((l, b_l), (u, b_u))) = ((exp^-(l), b_l), (exp^+(u), b_u))$

where both the min and max functions return a closed bound if the numerical values are identical and one of them is included in the interval.

These arithmetic evaluation rules can now be used within the evaluation step of the algorithm: for any constraint of the form $e_1 \bowtie e_2$, we can evaluate e_1 and e_2 inductively using our evaluation rules, having for each expression node, the result of evaluating the corresponding sub-expression. When both branches are evaluated, the resulting interval over-approximate the possible values of the underlying expression.

Interval Filtering Rules

The evaluation step of the algorithm computes over-approximations of the spaces defined by both branches of the constraint. The filtering step of the algorithm reduced these approximation to make them compatible according to the comparison operator. It requires to define filtering operation over to used representation. In order to be precise, these operations have to remove as much inconsistent values as possible without losing soundness. As this step follows the evaluation step and reuses its result, filtering operations can be seen as conditions to satisfy for the result.

Comparison Operators

First, we have to define filtering techniques for the comparison operators of the intervals. When dealing with an equality constraint, the corresponding operation on intervals is simply the intersection as we have to remove from both arguments all the values that can not be in the other argument. For the case of inequalities the filtering works as follows: given two intervals i, j, if we have $i \leq j$ we can remove from i all the values that are greater than the upper bound of j as they have no chance of being less or equal than any value in j. Symmetrically, we can remove from j all the values that are smaller than the lower bound of i for the same reason. The remaining rules for $\geq$ and $>$ can be easily deduced from the ones above.

Definition 41. *A filtering rule* $\leq_{filter}$: $B^2_{\mathbb{R}} \times B^2_{\mathbb{R}} \rightarrow B^2_{\mathbb{R}} \cup \emptyset \times B^2_{\mathbb{R}} \cup \emptyset$ *for the* $(\leq)$ *comparison operator is defined by: given two intervals* $i_1 = ((l_1, b_{l_1}), (u_1, b_{u_1}))$ *and* $i_2 = ((l_2, b_{l_2}), (u_2, b_{u_2}))$, $i_1 \leq_{filter} i_2$ *is equal to:*

$$(\emptyset, \emptyset) \text{ if } i_1 \cap i_2 = \emptyset$$

otherwise,

$$r = ((l_3, b_{l_3}), (u_3, b_{u_3})), \text{ where}$$

$$l_3 = min(l_1, l_2) \qquad u_3 = max(u_1, u_2)$$

$$b_{l_3} = \begin{cases} b_{l_1}, & if\, l_1 < l_2 \\ b_{l_2}, & if\, l_1 > l_2 \\ b_{l_1} \vee b_{l_2} & if\, l_1 = l_2 \end{cases} \qquad b_{u_3} = \begin{cases} b_{u_1}, & if\, u_1 > u_2 \\ b_{u_2}, & if\, u_1 < u_2 \\ b_{u_1} \vee b_{u_2} & if\, l_1 = l_2 \end{cases}$$

From the resulting filtered interval, we have to propagate the filtering towards the leafs of the tree. The idea is to define backward versions of the regular operators, that will enforce necessary conditions for the constraint to be respected. Given two interval arguments, we compute a subset of each argument by removing points that cannot satisfy the constraint. The result may also be empty in the case no point can satisfy the predicate.

Top-Down Filtering Rules

For each evaluation rule $\square_{eval}$, we have a corresponding filtering rule noted $\square_{filter}$. All of the filtering rules take one more parameter than their evaluation counterparts which corresponds to the constraint they should satisfy. For example, for a binary operator $\square$, we define $\square_{filter}$ which takes three parameters: the two parameters of $\square$ and the target result $a \square b$ have to stay in.

Definition 42. *Given a binary operator $\square$, a rule $\square_{filter} : B^2_{\mathbb{R}} \times B^2_{\mathbb{R}} \times B^2_{\mathbb{R}} \to (B^2_{\mathbb{R}} \times B^2_{\mathbb{R}}) \cup (\emptyset \times \emptyset)$ is a filtering rule for $\square$ if the equality $\square_{filter}(a, b, c) = (a', b')$ respects the following conditions:*

- *$a' \subseteq a \wedge b' \subseteq b$ (filtering)*
- *$\forall x \in a, x \notin a' \to \nexists y \in b, x \square y \in c$ (soundness 1)*
- *$\forall y \in b, y \notin b' \to \nexists x \in a, x \square y \in c$ (soundness 2)*

Here the filtering condition ensures that a rule only removes values from its arguments. Note that, in case the filtering operator empties an element, this information is passed to the other element in the product within $\square_{filter}$, hence, the situation where one element is empty and not the other one cannot happen. Both soundness conditions guarantees that if a value was removes, it was indeed inconsistent. Notice that a filtering condition does not need to remove all the inconsistent values. A filtering rule is considered complete if it respects the following property:

$$\square_{filter}(a, b, c) = c \to \forall x \in a, \forall y \in b, x \square y \in c$$

We can now illustrate the filtering rules over the same examples as before:

$$-_{filter}(a, r) = a \cap -_{eval}(r)$$
$$+_{filter}(a, b, r) = (a \cap -_{eval}(r, b)), (b \cap -_{eval}(r, a))$$
$$*_{filter}(a, b, r) = \left\{ \begin{array}{ll} (a \cap /_{eval}(r, b), b \cap /_{eval}(r, a)), & \text{for } 0 \notin a \wedge 0 \notin b \\ (a \cap /_{eval}(r, b), b), & \text{for } 0 \notin a \wedge 0 \in b \\ (a, b \cap /_{eval}(r, a)), & \text{for } 0 \in a \wedge 0 \notin b \\ (a, b), & \text{for } 0 \in a \wedge 0 \in b \end{array} \right\}$$

Here the filtering property is obviously respected, as all the results are obtained by intersection with the arguments. Moreover, the soundness condition is also respected as these operations are based upon evaluation rules as $a + b = r \implies b = r - a \wedge a = r - b$. As evaluation rules over-approximate the result, filtering rules behave similarly.

Thanks to these rules, we have now defined an evaluation and a filtering unit that work on continuous interval with included or excluded bounds that can be used within the *HC4* algorithm.

4.3.4 Exploration and Measurement

To meet the requirements of the solving method, we also need to define a split operation over this representation, and a way of measuring it.

Split Operation

We propose as an exploration method a simple bisection. Let $h = \frac{l+u}{2}$, we have:

$$\oplus(((l, b_l), (u, b_u))) = \{i_1, i_2\}$$

with:

$$i_1 = ((l, b_l), (h, true)) \tag{4.1}$$

$$i_2 = ((h, false), (u, b_u)) \tag{4.2}$$

Obviously, this split operator is correct as $\forall x \in \oplus(i), x \subseteq i$, hence no value are added by the split operation. This split is also complete as for all intervals i, $\oplus(i)$ forms a perfect cover of i. To preserve the irredundancy property of our solving method, we make sure that new bound of the obtained intervals h is contained by only them: For the real intervals, it should be closed for one of them and open for the other.

Of course, we can discuss other splitting operators, such being a splitting operator that forces monotonicity over the result of the split. As the Box abstract domain is built upon interval arithmetic, and operation in interval arithmetic are usually more expensive when the intervals do not have a constant sign (*e.g.* the multiplication of two interval requires to test the sign of the intervals, and then split them to reach monotonic interval, then join back the result), this allows the solver to be more efficient. This splitting operator operates in a way that allows it to obtain monotonic intervals when not already the case and gives good results in practice.

$$\oplus_{monotonic}(M) = \begin{cases} \oplus_{real}(M, 0) & \text{if } a_i \leq 0 \leq b_i \wedge m < i \leq n \\ \oplus_{mixed}(M) & \text{otherwise} \end{cases}$$

Size Function

The size function is simply defined as the difference between the upper bound and the lower bound, disregarding the fact that bounds may be open or closed. It is given by:

$$\tau((l, b_l), (u, b_u)) = u - l$$

This is compatible with the split operation, as splitting a (non-singleton) interval will always produces strictly smaller intervals according to τ.

4.4 Discrete Representation

Now that we have an appropriate representation for continuous domains, we are going to define one for domains that can take a discrete set of values. To represent integer ranges of values, we can use pairs of integer bounds which correspond to the (included) endpoints of the range.

Definition 43. *An integer interval is a pair (l, u), where $(l, u) \in \mathbb{Z}^2$, denoting the set of integer values x such that $l \leq x \wedge x \leq u$.*

This representation is very common and is well adapted to contiguous integer domains but may lose precision on some operation such as multiplication as shown in Example 14, where the result should be trivially even, but still, the resulting interval abstracts odd values.

Example 14. $(0, 10) * (2, 2) = (0, 20)$

One way to improve precision is to augment the interval representation with an abstraction that keeps track of the minimum distance between two consecutive values in the domain of a variable. In order to do so, we can go from the congruences abstract domain [Gra89] and make it CP-compatible. This representation is able to abstract infinite, discrete, equal-spaced set of integers with the notation $a\mathbb{Z} + b$, whose concretization is given by: $\gamma(a\mathbb{Z} + b) = \{ak + b | k \in \mathbb{Z}\}$ when $a \neq 0$ and is simply the singleton $\{b\}$ when $a = 0$. This abstract domain is meant to express properties of the form "the integer valued variable X is congruent to c modulo m", and is based on the partial order:

$$(a\mathbb{Z} + b) \sqsubseteq (a'\mathbb{Z} + b') \leftrightarrow a' \mid a \wedge b \equiv b' \pmod{a'}$$

Using this abstract domain in product with the integer intervals, we now obtain a discrete finite abstraction, able to express bounds for the possible values and the sparsity of the interval if need be.

In our case, since our domains are bounded, an element of such a product can be seen as a tuple (l, u, δ), where $(l, u) \in \mathbb{Z}^2$ and $\delta \in \mathbb{R}^>$, denoting the set of values $x \in \mathbb{R}$ such that $l \leq x \wedge x \leq u \wedge \exists k \in \mathbb{N}, x = l + \delta * k$.

Example 15 shows how this improved representation is able not to lose precision on the same multiplication.

Example 15. $(0, 10, 1) * (2, 2, 1) = (0, 20, 2)$

With this product, we can now abstract efficiently integer domains with congruences pattern, but when the space between values in smaller than one, this representation does not hold anymore.

In order to overcome this issue, we adapt this representation to any kind of discrete space and not only integers.

Definition 44. *Let $\mathbb{B}$ a computer representable subset of $\mathbb{R}$, d discrete interval is a tuple $(l, u, d) \in \mathbb{B}^3$, denoting the set of discrete values $x \in \mathbb{R}$ such that $l \leq x \wedge x \leq u \wedge \exists k \in \mathbb{N} = l + k * d$.*

This representation is a reduced product at a non-relational level of abstraction, and thus, the methods we have propose in the previous chapter also apply here to define the abstraction, concretization and set-theoretic operations also apply here. This is using the reduction operator ($reduce$) which given an interval $i = [l, u]$ and a congruence element $c = a\mathbb{B} + b$, reduces the interval in such a way that its bounds belong to $a\mathbb{B} + b$. It is not necessary to reduce the congruence element except in degenerate cases where the interval is a singleton or becomes empty after the reduction. It is such that reduce$(i, c) = [l', u']$ where:

$$l' = \begin{cases} l \text{ if } l \mod a = b \\ (l/a + 1) * a + b \text{ if } l \mod a \neq b \end{cases} \quad \text{and } u' = \begin{cases} u \text{ if } u \mod a = b \\ (u/a - 1) * a + b \text{ if } u \mod a \neq b \end{cases}$$

Hence we do not define all the operations but simply recall a few over the congruences for sake of concision, the original framework, over rational, being described in [Gra97].

4.4.1 Evaluation and Filtering

Let $d = a\mathbb{B} + b$ and $d' = a' + b'$ be two abstract values of the congruence domain. Then we define:

- $-_{eval} d = (a, -b)$
- $d +_{eval} d' = gcd(a, a')\mathbb{B} + (b + b')$

We don't give the proofs of these operators as they are easily lifted from [Gra92].

Comparison

As the operators $<, \leq, >\geq$ are not very useful over congruences, we can consider that they leave the abstract element unchanged. For the equality constraints, the filtering to be applied uses the meet:

$$=_{filter} (a, b) = (a \cap b), (b \cap a)$$

Top-Down Filtering Rules

The top-down filtering versions of the congruence operators follow the same principle than the one for intervals and we do not detail it here.

4.4.2 Exploration and Measurement

To meet the requirements of the solving method, we also need to define a split operation over this representation, and propose a way of measuring it.

Split Operation

Our discrete representation is a product of interval and congruences and as seen in Chapter 3, we can define several splits over a product representation. As we have already defined a split operator for intervals, we now define one for congruences. A natural split for congruences is to divide an element into two sub-elements, and to keep one value on two for the first, and the other for the second.

Definition 45. *Let $a\mathbb{Z} + b$ be a congruence abstract element, we define a split operator $\oplus$ such that:*

$$\oplus(a\mathbb{Z} + b) = \{a2\mathbb{Z} + b, a(2\mathbb{Z} + 1) + b\}$$

Proposition 4 (Congruence split)**.** *The congruence split operator is a correct split operator according to Definition 21.*

Proof. Let us prove that $\oplus$ is a finite, correct and complete split operator. We recall[4] that $\gamma(a\mathbb{Z}+b) = \{x | \exists k \in \mathbb{N}, x = a * k + b\}$.

- Trivially $\forall x, |\oplus(x)|$ is finite and equal to 2.
- We now prove that $\oplus$ is correct, *i.e.* $\forall x' \in \oplus(x), x' \sqsubseteq x$. In this case, we have to prove : $2 * a\mathbb{Z} + b \leq a\mathbb{Z} + b \wedge a(2\mathbb{Z}+1) + b \leq a\mathbb{Z} + b \Leftrightarrow \gamma(2 * a\mathbb{Z} + b) \subseteq \gamma(a\mathbb{Z}+b) \wedge \gamma(a(2\mathbb{Z}+1)+b) \subseteq \gamma(a\mathbb{Z}+b)$: First, we explicit that $\gamma(2a\mathbb{Z}+b) = \{x | \exists k \in \mathbb{Z}, x = 2 * a * k + b\}$. Also, we have: $\exists k \in \mathbb{Z}, x = 2 * a * k + b \rightarrow \exists k' \in \mathbb{Z}, x = a * k' + b$ with $k' = 2 * k$. Hence $\{x | \exists k \in \mathbb{N}, x = a * k + b\} \subseteq \{x | \exists k \in \mathbb{N}, x = 2 * a * k + b\}$. For similar reasons, we obtain that $\gamma(a(2\mathbb{Z}+1)+b) \subseteq \gamma(a\mathbb{Z}+b)$, which makes the $\oplus$ operator correct.
- Finally, we also prove that $\oplus$ is complete, *i.e.* $\forall e \in \gamma(x), \exists x' \in \oplus(x), e \in \gamma(x')$. In this case, we have to prove : $\forall x \in \gamma(a\mathbb{Z}, b), x \in \gamma(a2\mathbb{Z}+b) \vee x \in \gamma(a(2\mathbb{Z}+1)+b)$: First, we explicit that $\gamma(a\mathbb{Z}+b) = \{x | \exists k \in \mathbb{Z}, x = a * k + b\}$. By induction over $kmod2$:
  - case 0: $\exists k'$ such that $k = 2 * k'$, thus $x \in \gamma(a\mathbb{Z}, b) \rightarrow x \in \gamma(a2\mathbb{Z}+b)$
  - case 1: $\exists k'$ such that $k = 2 * k' + 1$, thus $x \in \gamma(a\mathbb{Z}, b) \rightarrow x \in \gamma(a(2\mathbb{Z}+1)+b)$

□

Moreover we have from the inductive reasoning of the proof of completion that $2 * a\mathbb{Z} + b \cap a(2\mathbb{Z}+1) + b = \emptyset$ which makes the $\oplus$ operator irredundant.

Also note here, we have made the choice of splitting an abstract element in only two sub-elements, but we can also define a more aggressive splitting operator $\oplus_k$ that would partition an abstract element into k sub-elements as following:

Definition 46. *Let $a\mathbb{Z}+b$ be a congruence abstract element, we define a split operator $\oplus_k$ such that:*

$$\oplus_k(a\mathbb{Z}+b) = \{k * (a\mathbb{Z}+i) + b | i \in [0; k-1]\}$$

From this definition we can adjust how much we want to split an abstract element and the previous definition is a particular of this one with $k = 2$.

As for the split defined in Definition 63, we can define a dispatch function that branches toward the congruence split or the interval split according to a set of constraints.

Size Function

As seen in Chapter 3, a size function over a product representation can be defined as the maximum value of the sizes of its components. We thus define a size function for the congruence domain. Defining a size function over infinite sets is not trivial and we will here focus on the space between two consecutive value of an element, disregarding the offset.

[4]operators are taken from [Min02]

Definition 47. *Let $d = a\mathbb{B} + b$ be an abstract element of the congruences domain, its size is given by:*

$$\tau(d) = 1/a$$

Which means, the more space we have between our elements the smaller the value will be considered. This size function is in accordance with the split defined in previous section as it respects for all abstract element of the congruences domain:

$$\forall x \in split(d), \tau(x) < \tau(d)$$

Still, in this case, it could be irrelevant to use the generic size function of the reduced product proposed in Section 3.4.2 as it is possible to compute the exact cardinality of a non-empty element of this product. Hence we propose a specialized size function:

Definition 48. *Let $id = [a, b], d$ be an abstract element of the congruences/interval reduced product, its size is given by:*

$$\tau(id) = (b - a)/d$$

This size function corresponds exactly to the number of elements of an abstract element.

4.5 For Mixed Integer-Real CSPs

We now have a representation that abstracts continuous set of values but behaves poorly on discrete sets, and one that is entirely dedicated to discrete values. To deal with mixed problems, *e.g.* problems with both integer and real variables, we define a mixed discrete-continuous abstract domain.

Definition 49. *Given a discrete set $\mathbb{S}_d$ and a continuous one $\mathbb{S}_c$, a mix representation is a member of the disjoint union:*

$$\mathbb{S}_d \uplus \mathbb{S}_c = \bigcup_{i \in \{d,c\}} \{(x, i) | x \in \mathbb{S}_i\}$$

The elements of the disjoint union are ordered pairs (x, i) where i serves as an auxiliary flag that indicates which S_i the element x came from.

We are now going to instantiate this mixed representation using both the continuous intervals and the discrete product of interval and congruences. We will then define arithmetic evaluation and filtering rules over this mixed representation that we will use inside the HC4 algorithm.

4.5.1 From the Order Theory Perspective

The mixed abstraction as a disjoint union of continuous intervals $\mathbb{C}$ and discrete interval/congruence products $\mathbb{D}$ is very handy as both of these sets feature a partial order (noted respectively $\leq_c$ and $\leq_d$) and conveniently, for all discrete interval/congruence product element $d \in \mathbb{D}$ there exists a smallest continuous interval $c \in \mathbb{C}$ such that c abstracts d, *i.e.*:

$$\forall c' \in \mathbb{D}, \forall x \in d, x \in c' \rightarrow c \subseteq c'$$

Given a discrete interval i with a and b bounds as lower and upper bounds, the corresponding smallest continuous interval $j \in \mathbb{D}$ is given by $((a, true), (b, true))$. Moreover, every element from i also belong to j. This makes our continuous representation a sound abstraction of the discrete one, which will allow us to define easily the operations over the mix representation given that we provide casting functions from to $\mathbb{D}$ to $\mathbb{C}$.

From this property, we can define a monotonic function *upcast* from $\mathbb{D}$ to $\mathbb{C}$:

$$upcast((a,b),d) = (a, true),(b, true)$$

This function respects:

$$\forall i, j \in \mathbb{D}, i \leq_d j \rightarrow upcast(i) \leq_c upcast(j)$$

We use this function to define an order over the disjoint union as follows:

Definition 50. *Let $\mathbb{M} = \mathbb{D} \uplus \mathbb{C}$. We can define a partial order over it as follows: $\forall x, y \in \mathbb{M}, x \leq y$ iff:*

$$\begin{cases} x \leq_d y, & \text{if } x \in \mathbb{D} \wedge y \in \mathbb{D} \\ x \leq_c y, & \text{if } x \in \mathbb{C} \wedge y \in \mathbb{C} \\ upcast(x) \leq_c y, & \text{if } x \in \mathbb{D} \wedge y \in \mathbb{C} \\ x = (a, true),(b, true) \wedge a = b \wedge a \in \gamma_d(y) & \text{if } x \in \mathbb{C} \wedge y \in \mathbb{D} \end{cases}$$

This order, which is build from the interval inclusion and the usual order over congruences is illustrated by the lattice on Figure 4.4. We use the notations $(a, b), d$ to designate a discrete interval with lower bound a and upper bound b and a distance d between each value of the domain, and we use the usual notation $[a, b]$, $[a, b[$... over continuous interval.

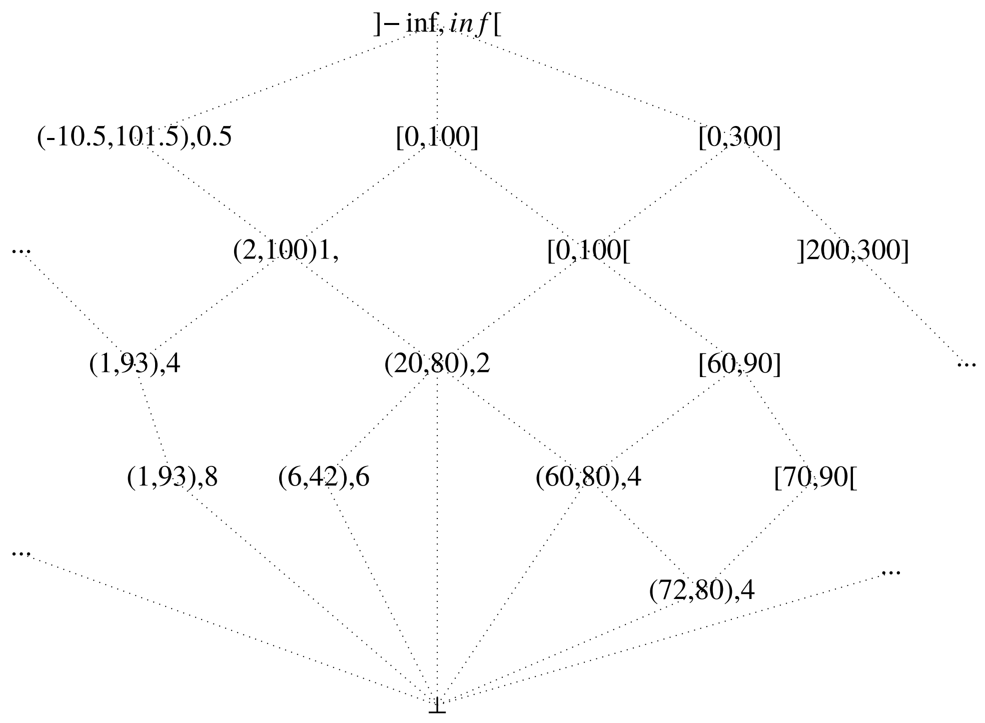


Figure 4.4: Lattice of mixed discrete/congruence interval and lattice of continuous intervals

From this lattice, we can explicit the meet and join relations: computing the intersection of two discrete intervals, (respectively continuous) yields a discrete (respectively continuous) interval. When meeting one discrete interval with a continuous one, in order to be precise, we can make the resulting interval discrete too[5] as is has to be included in its arguments and one of them is discrete.

Definition 51. *Consider the set $\mathbb{M} = \mathbb{D} \uplus \mathbb{C}$, we can define a meet operation over it $\cap : \mathbb{M} \to \mathbb{M} \to \mathbb{M} \cup \emptyset$: $x \cap y = z$ where z is equal to:*

$$\begin{cases} x \cap_d y, & \textit{when } x \in \mathbb{D} \wedge y \in \mathbb{D} \\ x \cap_c y, & \textit{when } x \in \mathbb{C} \wedge y \in \mathbb{C} \\ x \cap_d downcast(y) & \textit{when } x \in \mathbb{D} \wedge y \in \mathbb{C} \\ downcast(x) \cap_d y & \textit{when } x \in \mathbb{C} \wedge y \in \mathbb{D} \end{cases}$$

Trivially, when joining two discrete (respectively continuous) intervals, the result is also a discrete (respectively continuous) interval. When joining one discrete interval with a continuous one, in order to be sound, the resulting interval has to be continuous too.

Definition 52.

$$\begin{cases} x \cup_d y, & \textit{if } x \in \mathbb{D} \wedge y \in \mathbb{D} \\ x \cup_c y, & \textit{if } x \in \mathbb{C} \wedge y \in \mathbb{C} \\ x \cup upcast(y) & \textit{if } x \in \mathbb{C} \wedge y \in \mathbb{DI} \\ upcast(y) \cup y & \textit{if } x \in \mathbb{D} \wedge y \in \mathbb{C} \end{cases}$$

These join and meet relation can be seen as casting functions that allow our two representations to communicate. We use them to define evaluation and filtering rules.

4.5.2 Evaluation and Filtering

We exploit the fact that an arbitrary expression defines a space that is either discrete or continuous, to refine our evaluation and filtering rules using the mixed abstraction. We augment our rules to define the behaviour our the arithmetic when an operation is applied on two argument that have a different continuity. Figure 4.5 illustrates the behaviour of the addition on this mixed representation.

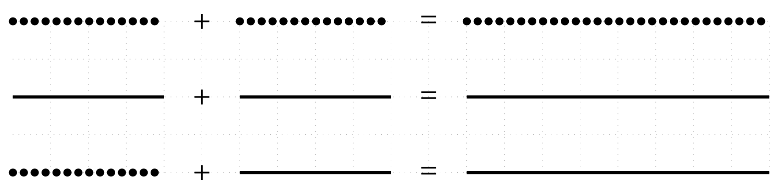


Figure 4.5: Concrete spaces define by addition of intervals according to their continuity. Dotted segment denote discrete intervals and plain ones continuous intervals

In all the evaluation operations, whenever we can not guarantee the discrete property of the result, we make it continuous as it is a sound over-approximation. In the way, when our filtering rules is able to infer the discrete nature of the result, we regain precision and make it discrete.

[5] even though its correct to make it continuous as all of our functions do not need to be exact and can compute an over-approximation

4.5.3 Exploration and Measurement

Both of the exploration and the measurement operations lifts naturally to the union representation as it consists in a simple dispatch.

Size Function

The precision function corresponds to the size of the largest dimension:

$$\tau_m(\{a_1, b_1\} \times \ldots \times \{a_m, b_m\} \times [a_{m+1}, b_{m+1}] \times \ldots \times [a_n, b_n]) = \max_i(b_i - a_i)$$

Choosing a $r < 1$ guarantees that all the integer variables are instantiated when the solvers terminates.

Split Operator

We now give the split operator for our unified representation, which is a simple dispatch according to the continuity: If v is a variable with a finite number of values then we use the discrete split operator $\oplus_{dis}$ and we use the continuous split operator $\oplus_{con}$ otherwise, where $\oplus_{dis}$ and $\oplus_{con}$ are given respectively by:

Definition 53. *Let v be the variable we want to split, and r its associated abstraction. The split operator* $\oplus$ *is defined as follows:*

$$\oplus(r) = \begin{cases} \oplus_{\mathbb{D}}(r) \textit{ if } r \in \mathbb{D} \\ \oplus_{\mathbb{C}}(r) \textit{ if } r \in \mathbb{C} \end{cases}$$

Proposition 5 (Mixed split)**.** *The mixed discrete-continuous split operator is a correct, complete split operator according to Definition 21.*

Proof. $\oplus_{dis}$ and $\oplus_{con}$ are both correct, complete, and produce a finite number of elements. Trivially, as the mixed split branches either in $\oplus_{dis}$ or $\oplus_{con}$, it is also correct complete and produce a finite number of elements. □

4.6 Discrete-Continuous HC4

Thanks to the mix discrete-continuous representation, we are now able to use the bottom-up top-down procedure with a more expressive non-relational abstract domain. This gain of precision is more interesting than using an integrity constraint on integer variables as it generalizes to any discrete set of values, not only integers and it is able to detect inconsistencies sooner.

For example if we consider the Problem 16, we can see that the left side $x/2$ of the constraint evaluates to the discrete abstract value $[0; 5], 0.5$ and the right side evaluates to the continuous interval $[0; 3]$. After propagation of the $(=)$ operator, we obtain for both sides the discrete element $[0; 3], 0.5$, and the variables are updated accordingly: x is any integer in $[0; 6]$ and y is any real in $[0; 1]$ of the form $(1/6)k$. In one round of propagation, we have found all the solution of the constraint. If we were using a filtering of the integer variables, it would have not be the case, as only the variable x would have been filtered. Moreover, several stages of exploration would have been necessary to divide the variable x and enumerate it to obtain the same level of precision, without counting the potential split of the variable y.

Example 16. *An example with one real variable y and one integer variable x, subject to 1 constraint.*

- $x \in \mathbb{N} \wedge x \in [0; 10]$
- $y \in \mathbb{R} \wedge y \in [0; 1]$

Constrained with:

- $x/2 = 3 * y$

4.7 Experiments

We now discuss the implementation and performance aspects of this work. We have implemented the mixed discrete-continuous abstract domain inside the AbSolute solver and measure its performance over the minlp benchmark which involves both problems with continuous and discrete variables. We have selected problems with both integer and real variables. We compare the resolution times obtained with our mixed discrete continuous abstract domain with a simple technique consisting in enforcing an integrity constraint on integer variables when one is encountered.

4.7.1 Analysis

The abstract domain we have developped maintains a more expensive representation than a simple interval, and the computations we perform with it are naturally more costly. However, we are still able to outperform the integrity constraint-based strategy with it. 16 over 22 examples where solved faster using our mixed representation, sometimes improving the resolution time by a factor of 2. This is due to the fact that our improved precision allows the solver to maintain a more precise abstraction and inconsistencies are detected sooner. This early detection results in a lot of computations being avoided. This confirms our intuition that a mixed representation is more interesting than a fully-continuous one with additional processing for discrete variables. Also, even when no discrete variable is present in a problem, a discrete behaviour may still appear due to the constraints, or the constants of the problem, in which case our method can still apply counter to the integrity constraint-base technique.

4.8 Conclusion

4.8.1 Contributions

The abstractions we have defined in this chapter are more general than the usual interval representation used in continuous solving. For the discrete spaces, it also subsumes the bound consistency as they do not only keep the lower and upper bounds for each variable but also the minimum “distance” between two consecutive possible values of a variable. This representation, along with the arithmetic and filtering procedures we have defined on it is able to deal with both discrete representations as well as continuous ones, while minimizing the loss of precision and maintaining efficiency. It goes beyond the classic integrity constraint applied on integer variables, as it can express any kind of discrete sets, and not only integers (*e.g.* $\{0.5, 1, 1.5, 2...\}$), and is able to detect inconsistencies much sooner as it is more precise.

Table 4.1: Comparing the mixed discrete continuous abstract domain with the filtering of discrete variables.

problem	#var	#ctrs	time, Mixed	time, Filter
booth	2	2	**4.338**	6.36
descartesfolium	2	2	**0.002**	0.113
cubic	2	2	**0.003**	0.006
parabola	2	2	**0.002**	0.008
precondk	2	2	0.014	**0.004**
exnewton	2	3	**0.750**	1.342
supersim	2	3	0.012	**0.008**
zecevic	2	3	19.583	**17.382**
hs23	2	6	1.982	6.239
aljazzaf	3	2	**0.008**	0.021
bronstein	3	3	0.608	**0.401**
eqlin	3	3	0.073	**0.008**
kear12	3	3	**0.491**	0.601
powell	4	4	**0.143**	0.321
h72	4	0	**0.007**	0.012
vrahatis	9	9	**0.062**	0.115
dccircuit	9	11	**0.188**	0.456
i2	10	10	0.107	**0.100**
i5	10	10	**0.026**	0.211
combustion	10	10	**0.003**	0.011
st_miqp5	7	15	**1.661**	2.507
hs5	2	1	**1.672**	3.498
supersim	2	3	**7.116**	14.673

Our implementation and preliminary results are encouraging and give a good hope concerning the design of a generic constraint solver, able to mix different representation, either discrete or continuous, transparently and efficiently.

4.8.2 Perspectives

Now that we are able to handle discrete variables, further work could concern the incorporation of standard heuristics over the discrete inside our solver.This opens the possibility of new real-world applications as it is common for applications requiring the use of continuous variables to also use discrete variables. For example, layout problems are often parameterized by an integer number of containers, or one of the moving axes of a robot moves continuously but with an integer number of tasks to perform. Also this work can be used as a starting point for the design of other non relational abstract domains for Constraint Programming, for example a powerset representation can be an interesting addition to the AbSolute solver, on the same principle than the product we used.

Chapter 5

Propagation with Elimination

Abstract

In continuous constraint programming, the solving process alternates propagation steps, which reduce the search space according to the constraints, and branching steps. In practice, the solvers spend a lot of computation time in propagation to separate feasible and unfeasible parts of the search space. The constraint propagators cut the search space into two sub-spaces: the inconsistent one, which can be discarded, and the consistent one, which may contain solutions and where the search continues. The status of all this consistent subspace is thus indeterminate. In this chapter, we introduce a technique called *elimination*. It refines the analysis of the consistent subspace by dividing it in a more relevant way than a standard split. Elimination relies on the propagation of the negation of the constraints, and a new difference operator to efficiently compute the obtained set as an union of boxes, thus it uses the same representations and algorithms as those already existing in the solvers. Combined with propagation, elimination allows the solver to focus on the frontiers of the constraints, which is the core difficult part of the problem. We have implemented this method in the AbSolute solver, within all of our abstract domains, and present experimental results on classic benchmarks with good performances.

Contents

5.1 Motivation and Related Works

Solving a constraint satisfaction problem on a finite domain is a NP-complete problem in the general case. Constraint solvers generally rely on propagation to filter the domains and exploration to build "easier" sub-problems. The efficiency of a solver depends on the choices made by the exploration process, an these choices are often guided by heuristics. Depending on the type of constraint, and type of variables, these heuristics vary a lot. On discrete variables, such heuristics can for example try and provoke early failures (such as *fail-first* [HE79] or *dom /w deg* [BHLS04]). On continuous variables, classic heuristics include: *largest first* [Rat94], which consists in splitting the largest domain; *round robin*, where the domains are processed successively; or *maximal smear* [Han92], choosing the domain with the greatest slope based on the derivatives of the constraints. Also, even when the variable is fixed, the way the domains are splitted can vary: for example in [ZMRM17], domain splitting strategies involve simple bisection 3-way, 5-way and 6-way split, where interval floating point domains are splitted in two intervals but some points of interest are put aside to be handled separately. Closer to the work we present in this chapter, [DCM08, Vio07] and [KB05] whose principles are based on the discovery and memorization of associations variable/values that lead to infeasible instances. It is therefore a learning system that enriches the knowledge of the solver about the problem and increases its efficiency as it progresses.

In this chapter, we propose to add a new resolution step, also based on the infeasible region of the search space, but no learning mechanism. This step is complementary to constraint propagation, and postpones the exploration step, we call it the *elimination* step. This step divides the search space into three sub-spaces: one containing only solutions, one containing only inconsistent instanciations and the last one where the constraints are indeterminate — it may contain solutions as well as non-solutions. Our solving method will thus alternates three steps: propagation, elimination, and exploration. It offers another way of reasoning on the constraints, since we are not only exploiting the constraints' consistencies (as does propagation) but also the constraint inconsistencies. With this improved reasoning, the interesting zones of the search space are better targeted: zones without solutions are discarded by propagation, and zones with only solutions are set aside by elimination into the solution space, which means in practice that they also are excluded from the search. The search effort can then focus on the indeterminate space — the part of the search space effectively requiring deeper exploration by the solver. For sake of clarity, we will first illustrate the method within the interval abstract domain, before generalizing to any abstract domain and defining the necessary operators for polyhedra, the reduced product abstract domain, and the mix discrete-continuous representation.

5.2 Limitations of a Propagation-Exploration Loop

Firstly, we illustrate how a propagation exploration loop can be maladjusted in some cases, by showing its results over the following example:

Example 17. *An example with 2 variables $(x, y) \in \mathbb{R}^2$ subject to 1 constraint.*

- $x1 \in [1; 50]$
- $x2 \in [-1.5; 1]$

Constrained with:

$\cos(\ln(x)) > y$

Resolution:

Abstract domain used	*Box*
Inner solutions	*3717*
Inner volume	*36.73*
Outer solutions	*2048*
Outer volume	*0.94*
Inner ratio	*97.49*
Solving time	*0.04s*
Precision	*0.03*

Figure 5.1 shows the result obtained with Algorithm 5, for the problem 17.

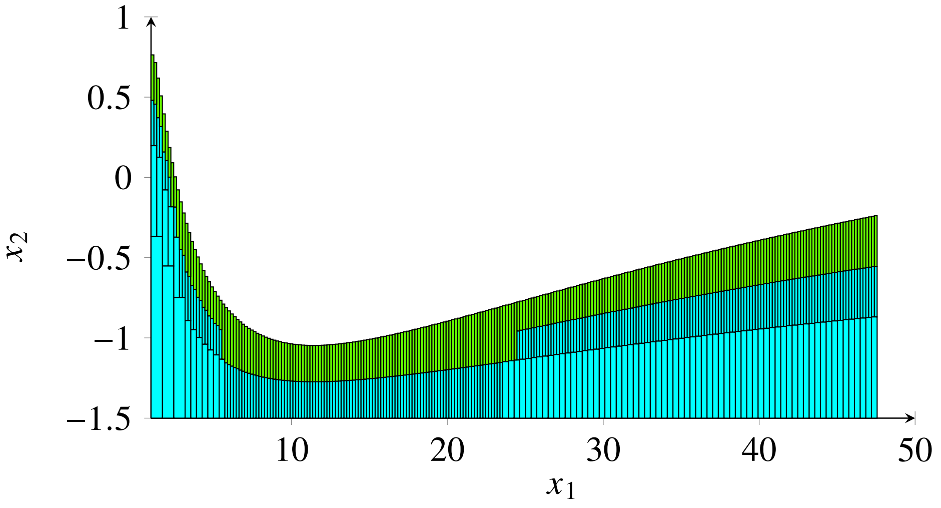


Figure 5.1: Output of AbSolute using the Box abstract domain, with precision set 0.3, *largest-first* as a split heuristic and an unbounded number of iteration

Using the *largest-first* heuristic induces the solving process goes through a lot of exploration step, some of which are irrelevant. In this case, one of the variable, $x1$, has a much larger domain than the other, which causes the exploration to focus mainly on this variable. Using another variable selection strategy (*e.g. first-fail*) would not settle this issue as it would also produce unnecessary split but on the $x2$ variable. Figure 5.1 shows that some of the inner elements could be merged to obtain fewer, but larger inner boxes. This observation reflects the fact that some exploration steps are unnecessary.

Remark The neediness of an exploration step is not a clearly defined measure, but here, we will consider that if an exploration step produces a set of element that we could merge together at the end of the solving process, without loosing the correctness property, than this step is unnecessary.

This has a result the production of a cover composed of a large number of boxes, which can make it impractical for a reuse. The elimination step we introduce in the next section pushes the reasoning based on the constraints one step further, before branching to the exploration, in order to avoid these superfluous steps.

5.3 Another Point of View

A concrete instanciation either satisfy a constraint, or its negation. This is not the case for abstract element as an abstraction function may induce approximation and loss of precision.

The propagation step reduces the search space by removing non-consistent sub-spaces. Elimination aims at reducing the search space by focusing on the frontiers of the problem. This is done in three steps: computing the elimination for each constraint, combining the result with the domains with a new difference operator, and finally integrate this mechanism in the solving process.

We firstly introduce the concept of elimination for a single constraint. It relies on the constraint propagator to over-approximate the set of instanciations that *can not* be solutions. We will refer to these instanciations as inconsistent instanciations. By elimination, the rest of the search space can only contain solutions.

Let $\mathcal{V} = x_1, \ldots, x_n$ be variables in domains $\mathcal{D} = d_1, \ldots, d_n$, and c a constraint on $x_1, \ldots, x_n$. We define a function $\theta_c : \mathcal{B} \to \mathcal{B}$ such that $\theta_c(d_1, \ldots, d_n) = \rho_{\neg C}(d_1, \ldots, d_n)$. It can be seen as a complementary propagation step. Combining this function with propagation, we can partition the search space relatively to the satisfiability of the constraint c. Let $S_C = \rho_C(d_1, \ldots, d_n)$, an over-approximation of $Sol(< \mathcal{X}, \mathcal{D},, \{c\} >)$ and $\overline{S_C} = \theta_C(d_1, \ldots, d_n)$ an over-approximation of $\overline{Sol}(< \mathcal{X}, \mathcal{D}, \{c\} >)$. We can differentiate from this partitioning three kinds of instanciations:

- the instanciations that belong to $\overline{S_c}$ and not to S_c: they are guaranteed to be inconsistent and can be discarded from the search-space.

$$\overline{S_c} \setminus S_c \cap Sol(< \mathcal{X}, \mathcal{D}, \{c\} = \emptyset$$

- the instanciations that belong to S_c, and not to $\overline{S_c}$: they are guaranteed to be consistent and can also be discarded from the search-space to be added into the solution set.

$$S_c \setminus \overline{S_C} \subseteq Sol(< \mathcal{X}, \mathcal{D}, \{c\}$$

- the remaining instanciations that belong to both S_C and $\overline{S_C}$: they are indeterminate.

This form of reasoning can be seen as a new contraction, in the same framework as the contractors described in [JW93] and used in Ibex [CJ09] to perform a smarter exploration. We add an automatic propagation on the negation of the constraints, to identify sub-spaces containing only solutions. We thus reason on the negation of the constraints, hence we compute sets which are not boxes: to overcome this issue, we also add an operator on boxes to efficiently compute the difference of two boxes (or the relative complementary of one box in another) as a union of boxes. Thus, our method can be integrated into any solver without changing its domain representation nor modifying the propagators.

Figure 5.2 shows an example of this partitioning. For the constraint $y \leq x^3$ (filled with blue), the box $\mathcal{S}_C$ (dashed), computed through propagation, over-approximates the solutions and the box Θ (hatched in green), computed by applying propagation over the negation of the constraint, over-approximates the inconsistencies. We can see that the complement of Θ under-approximates the set of solutions, while the complement of R under-approximates the set of inconsistencies. The intersection $\Theta \cap R$ can contain both solutions and inconsistencies.

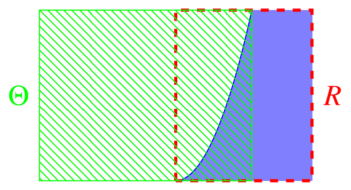


Figure 5.2: Given the constraint $y \leq x^3$, in blue, the box R over-approximates the solutions and the hatched box Θ over-approximates the inconsistencies.

Once this partitioning is done, the inconsistent part can be discarded (as usual) and the consistent one can be directly added to the set `sols` of solutions. What remains is the indeterminate space in which the solving process continues. This principle is then generalized to the case of several constraints: the consistent part is the intersection of all the consistent parts associated to each constraint. Symmetrically, the inconsistent part is the union of all the inconsistent parts associated to each constraint. What remains is the indeterminate part.

Remark In practice, in the case of continuous constraints, elimination can rely on the original propagation algorithms of the considered constraint, since we can easily compute the negation of a constraint (based on predicates $<, =, \leq$). It would also be valid for discrete constraints *provided that* the same property holds. Indeed, primitive constraints could be dealt with elimination, but handling global constraints would require to specifically define their negations and introduce dedicated propagators. Constraints languages based on atomic constraints could be dealt with our framework (for instance the primitive constraints of CLP(FD), *i.e* the arithmetic constraints, but not global constraints: the propagation of the negation of a global constraint (for instance, **all_different**) cannot be done with the same algorithm as the global constraint (in our example, this would require another global constraint **at_least_two_equal** to express the negation of **all_different**, with a specific propagator).

The indeterminate space is defined as an intersection of boxes, which results in a box. Hence, the solving process continues within a box, as in a classic propagation-based solver, except that the box is possibly smaller as we intersect the result of propagation with the result of elimination. However, $S_C \setminus \overline{S_C}$ is not necessarily a box. Computing this set difference requires taking the complement of a box relative to another box. In the following section, we define a set difference operator over boxes. It computes the difference as a set of boxes, that can be directly added to `sols`.

5.4 Difference Operator

The abstract domain used classically in Abstract Interpretation are generally not closed under difference. For example, given two boxes B_1 and B_2, their difference $B_1 \setminus B_2$ is not necessarily a box. However, we can express it as a collection of boxes that covers $B_1 \setminus B_2$. A cover is sufficient to have a sound and complete resolution method, and is easier to build as we will see in the current section. Our difference operator should thus satisfy the following properties:

Definition 54 (Difference operator)**.** *A difference operator $\ominus : \mathcal{B} \times \mathcal{B} \to P(\mathcal{B})$ is a binary operator such that $\forall B_1, B_2 \in \mathcal{B}$:*

1. *$|B_1 \ominus B_2|$ is finite;*

2. *$\forall b \in (B_1 \ominus B_2) \Rightarrow b \cap B_2 = \emptyset$;*

3. *$B_1 = (B_1 \cap B_2) \cup \bigcup \{ b \in B_1 \ominus B_2 \}$.*

The first condition ensures that the solving method produces a finite set of boxes. The second one ensures that the operator eliminates from the box B_1 the values inside the box B_2. Finally, the third condition guarantees that the difference of B_1 and B_2, union B_2, covers the initial box B_1. The second condition is related to soundness and the third one to completeness.

Our difference operator on boxes uses the constraint conjunction representation. A box can be defined as a conjunction of constraints $B = \bigwedge_{i=1,\dots,p} c_i$, where each constraint $c_i = \pm x_i \triangleleft a_i$, with $\triangleleft \in \{<, \leq\}$, gives a lower or an upper bound — not necessarily included — on x_i. Using the constraint representation allows our difference operator to handle potentially unbounded boxes.

Note that it is mandatory to be able to express both strict and large inequalities. Otherwise, a problem would arise as the negation of $\pm x_i > a_i$ would not be exactly representable, and we would have no way to ensure property Def. 54.2. As the difference operator is used to compute $\mathcal{S}$, an under-approximation of the set of solutions, adding to $\mathcal{S}$ the closure of boxes which should actually be open, could add to it points that are not solutions to the problem, and thus break the soundness criterion.

Each c_i defines a half-space, and the intersection between a box and a half-space is still a box[1]. A first step is thus to compute the difference between two boxes, by considering each half space of the box to remove independently.

Definition 55 (Difference for boxes)**.** *Let B_1 and B_2 be two boxes, with B_2 represented as the set of constraints C_2. The difference of B_1 and B_2 is:*

$$B_1 \ominus B_2 \triangleq \{B_1 \cap (\neg c) \mid c \in C_2\} \tag{5.1}$$

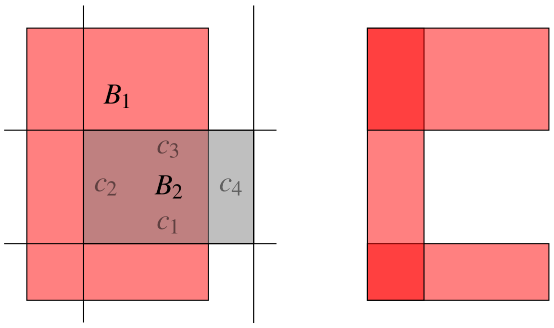


Figure 5.3: Result of a difference of two boxes : $B_1 - B_2$

This naive method can result in widely overlapping boxes in the output. Nevertheless, it is an acceptable difference operator as it satisfies Def. 54.

Proposition 6. *$\ominus$ is valid difference operator according to definition 54.*

Proof. Let us prove that $\ominus$ is a finite, correct and complete difference operator.

[1]as the half-space are parallel to the variables axis.

- Obviously $\forall B_1, B_2, |B_1 \ominus B_2|$ is finite and equal to number of constraint of B_2, $|C_2|$.

- Let $C_2 = \{c_1 \wedge c_2 \wedge \ldots \wedge c_n\}$, the correction property come simply from the identity: $B_1 \setminus B_2 = B_1 \cap B_2^C$ and as $B_2^C = \{c_1 \wedge c_2 \wedge \ldots \wedge c_n\}^C$ can be expressed as $\{\neg c_1 \vee \neg c_2 \vee \ldots \vee \neg c_n\}$ thanks to De Morgan's laws, we have trivially that $B_1 \setminus B_2 = B_1 \cap \{\neg c_1 \vee \neg c_2 \vee \ldots \vee \neg c_n\}$ which rewrites in our definition $\{B_1 \cap (\neg c) \mid c \in C_2\}$ thanks to the distributivity of the conjunction over the disjunction. As this intersection does not loose precision, $B_1 \setminus B_2 = B_1 \cap B_2^C$ has an empty intersection with B_2 by definition of the complementation.

- We now show that we do not loose any instance of B_1 that could potentially be solution, *i.e.* $B_1 = (B_1 \cap B_2) \cup \bigcup \{ b \in B_1 \ominus B_2 \}$. By definition of the complement, B_2, B_2^C is a partition of any subset, in particular B_1. Hence B_1 is entirely covered by $B_1 \cap B_2$ and by $B_1 \setminus B_2$ and we do not loose solutions.

□

As a side note, we can state from the proof 5.4 that we can define a generic difference operator over any abstract domain closed by intersection given that we can express the complementary of an abstract element exactly, as a powerset of elements for example.

Non-Redundant Difference Operator

We are now able to cover the difference of two boxes, without loosing the soundness. However, to guarantee a non-redundancy property over the result, this cover should be made a partition. This would prevent boxes from overlapping and have instanciations covered by several elements. We then strengthen our definition with the following property:

Figure 5.3 shows an example of the application of the difference operator on two boxes. The left-side of the figure gives the initial boxes B_1 and B_2, with B_2 represented by the constraints $\{c_1, \ldots, c_4\}$. The right-side of the figure shows the result of the naive difference operator. Here, $B_1 \setminus B_2$ is covered by three elements, one per constraint of C_2, after removing the constraints that, intersected with B_1, yield the empty set (c_4 in this case). Overlapping boxes in the output appear in a darker shades. This overlapping implies that some instanciations may be covered by more than one box: the result is redundant.

We now propose an improved difference operator in order to obtain non-overlapping boxes when building a partition of $B_1 \setminus B_2$.

5.4.1 Toward Irredundancy

Definition 56 (Non-redundant difference operator)**.** *A non-redundant difference operator is a difference operator with respect to Definition 54, and respects furthermore the following property:*

1. $\forall x, y \in (B_1 \ominus B_2), x \cap y = \emptyset$

We enforce in this strengthened definition that no pair of element of the result intersect to maintain an irredundant solving method.

Definition 57 (Non-redundant difference for boxes)**.** *Let B_1 and B_2 be two boxes and B_2 is represented by the set of constraints $C_2 = \{c_1, \ldots, c_p\}$. The difference of B_1 and B_2 is defined as:*

$$B_1 \ominus_{irr} B_2 \triangleq \left\{ B_1 \cap (\neg c_i) \cap \bigcap_{j<i} c_j \mid i \in \{1, \ldots, p\} \right\} \tag{5.2}$$

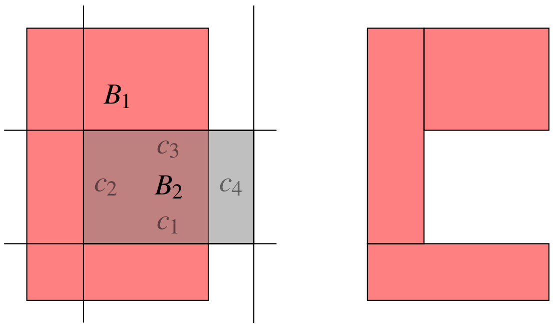


Figure 5.4: Result of a non redundant difference of two boxes : $B_1 - B_2$

For similar reasons to the naive difference operator, Def. 54.1–3 is also satisfied for the non-redundant difference operator. Additionally, we strengthen the property that $B_1 \ominus_{irr} B_2$ is a cover for $B_1 \setminus B_2$ by making this cover a partition, i.e, the elements of $B_1 \ominus_{irr} B_2$ are pairwise disjoints: we ensure that, for any pair of boxes $b_i, b_j \in B_1 \ominus_{irr} B_2$ such that $i \neq j$, we have $b_i \cap b_j = \emptyset$.

Proposition 7. *$\ominus_{irr}$ is a valid difference operator.*

Proof. $\ominus_{irr}$ is a valid difference operator for the same reasons than the redundant operator $\ominus$. □

Proposition 8. *$\ominus_{irr}$ is an irredundant difference operator.*

Proof. If $|B_1 \ominus B_2| = 1$ then, trivially, $B_1 \ominus B_2$ is a partition of $B_1 \setminus B_2$. If $|B_1 \ominus B_2| > 1$, we have to prove that the elements of $B_1 \ominus B_2$ are pairwise disjoints. Let $C_2 = \{c_1, ..., c_p\}$ be the constraints of B_2, and b_i, b_j be respectively the i-th and the j-th value of $B_1 \ominus B_2$ according to (5.2), with $i, j \in 1..p$ and $i \neq j$. Then, b_i is constrained by $\neg c_i$. Assuming w.l.o.g. that $i < j$, then b_j is constrained by c_i, and $b_i \cap b_j = \emptyset$. We also have to prove that $B_1 = B_1 \setminus B_2 \cup B_2$, or equivalently, $\cup_i b_i = B_1 \setminus B_2$: let $x \in \cup_i b_i$ be an instanciation of B_1. By definition of B_2, there is at least a constraint $c_i \in C_2$ such that x does not satisfy. Let i_0 be the smallest such i, then $x \in b_{i_0}$. Thus, the whole of $B_1 \setminus B_2$ is covered by the boxes b_i. □

5.4.2 Difference Operator for Other Abstract Domains

As our solving method can be used within any abstract domain, we show how to define a difference operator for the other representations we use to have the elimination technique enabled with those.

Difference operator for octagons and polyhedra

Interestingly enough, the definition of the difference operator holds for any kind of linear constraint conjunction, as long as the representation is closed under intersection and complementation (as the difference $A \setminus B$ can be rewritten in $A \cap B^C$). Given that the domains feature both strict and large

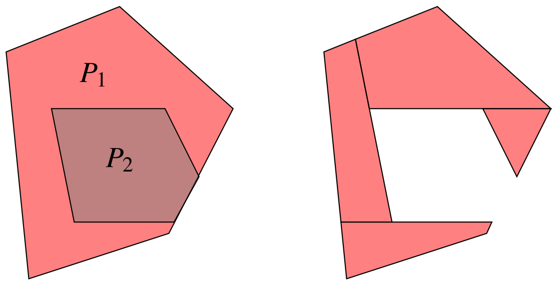


Figure 5.5: Difference operator for polyhedra

constraints, our definition holds also for the polyhedra abstract domain as exemplified by Figure5.5 and the octagon[2] abstract domain. Figure 5.5 illustrates an application of the non-redundant difference operator over polyhedra.

We omit the formal definition and proofs of this operator with polyhedra, as the ones we gave for the interval abstract domain manipulate conjunction of linear constraint and can be reused for polyhedra.

Difference operator for Reduced Product

In chapter 3, we defined domain products to exploit the expressivity of several domains. We also define a difference operation for these to make the elimination available with any combination of domain products.

Definition 58. *Let $(A \times B)$ and $(C \times D)'$ be two abstract elements of the reduced product abstract domain and D_1 and D_2 the underlying abstract domains with $\ominus_{D_1}$ and $\ominus_{D_2}$ their correct, complete and irredundant difference operators. The difference operator $\ominus$ over the reduced product $D_1 \times D_2$ is given by:*

$$(A \times C) \ominus (B \times D) = \{(x, y) | x \in A \ominus_{D_1} C \wedge y \in B \ominus_{D_2} D\}$$

Proposition 9 (Product difference)**.** *The product difference operator is a correct difference operator according to Definition 56.*

Proof. Let us prove that $\ominus$ is a finite, correct and complete difference operator. As seen previously $(A \times B) \ominus (C \times D)$ is equivalent to $(A \times B) \cap (C \times D)^C$. As a product is a conjunction, we can use De Morgan's laws:

$$\begin{aligned}(A \times B) \ominus (BC \times D) &= (A \times B) \cap (C^C \cup B^C)\\ &= (A \cap C^C) \cap (B \cap D^C)\\ &= (A \ominus_{D_1} C) \cap (B \ominus_{D_2} D)\end{aligned}$$

Since both $\ominus_{D_1}$ and $\ominus_{D_2}$ are correct, complete and irredundant difference operators from hypotheses, $\ominus$ is a finite, correct and complete difference operator. □

[2]The standard implementations of the octagon abstract domain (*e.g.* Apron's octagons) are designed for program verification purposes and generally feature only large constraints. This is due to the fact that the spaces they have to abstract are discrete as they correspond to either floating point or integer spaces. This makes the use of those implementations inappropriate with the elimination technique.

Difference Operator for the Mixed Discrete-Continuous Representation

In chapter 4, we defined abstractions for mixed discrete-continuous spaces, and we now show the the elimination step can also be used within these abstractions. All we have to do is to define a difference operator for it. We have that this representation corresponds to a product and we have just defined and elimination operator over products. Hence, all we have to do is to define an difference operator over congruences and lift it to the product representation. To do that, we can notice that $A \setminus B$ is equivalent to $A \setminus (A \cap B)$ as the set difference operation corresponds to the set of elements that belong to first operand but not to the second. This allows us to handle only the cases where the second operand is included in the first one.

Definition 59. *Let $a\mathbb{Z} + b$ and $a'\mathbb{Z} + b'$ be two abstract elements of the congruence abstract domain, with $a'\mathbb{Z} + b' \leq a\mathbb{Z} + b$. The difference operator $\ominus$ over the congruences is such that: let $k = a'/a$, and $k = k_0 * k_1 * \cdots * k_n$ its integer prime factorization in increasing order. Let $\{c_0, c_1, \ldots, c_m\} = \oplus_{k_0}(a\mathbb{Z}+b)$, we have:*

$$(a\mathbb{Z} + b) \ominus (a'\mathbb{Z} + b') = \left\{ \begin{array}{ll} \emptyset, & \textit{if } k = 1 \\ \{c_{k-1}, c_{k+1}, \ldots, c_m | a * k = a'\} \cup (c_k \ominus (a'\mathbb{Z} + b')) & \textit{if } k > 1 \end{array} \right\}$$

Intuitively, when computing $a \ominus b$, we go from the bigger value a toward the bottom of the lattice, splitting it successively and collecting the values that do not include b. We explicit an application of this difference operator over the following example, illustrated by figure 5.6:

Example 18. *Having* $36/3 = 12 = 2 * 2 * 3$, *we obtain :*

$$3\mathbb{Z} + 1 \ominus 36\mathbb{Z} + 27 = \{6\mathbb{Z} + 4, 12\mathbb{Z} + 1, 36\mathbb{Z} + 7, 36\mathbb{Z} + 15\}$$

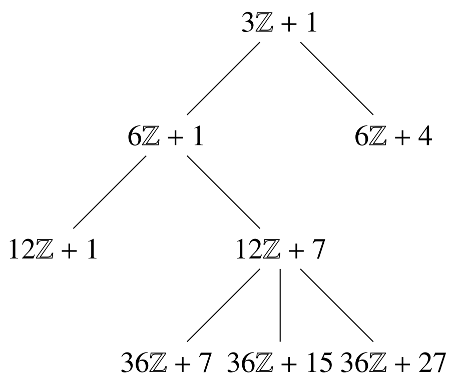


Figure 5.6: Decomposition of $3\mathbb{Z} + 1$ to compute the difference with $36\mathbb{Z} + 15$

Proposition 10 (Mixed discrete-continuous difference)**.** *The congruence difference operator is a correct difference operator according to Definition 56.*

Proof. Let us prove that $\ominus$ is a finite, correct and complete difference operator.

- when $a = a'$, we have $a'/a = 1$ and trivially $a\mathbb{Z} + b \ominus a'\mathbb{Z} + b' = \emptyset$, which is correct and complete.

- When $a < a'$, let $k_0, k_1\ dotsk_n$ prime factorization of $k = a'/a$. As for all k_i, we have a' can be divided by $a * k_i$, by construction, we have the identity:

$$\forall k_i, (a\mathbb{Z} + b) \ominus (a'\mathbb{Z} + b') = \bigcup x, x \in \oplus_{k_i}(a\mathbb{Z} + b) \ominus (a'\mathbb{Z} + b')$$

  By construction, $\exists c \in \oplus_{k_i}(a\mathbb{Z}+b)$ such that $a'\mathbb{Z}+b' \leq c$. Then by associativity of the disjunction we have:

$$(a\mathbb{Z} + b) \ominus (a'\mathbb{Z} + b') = (\oplus_k(a\mathbb{Z} + b) \setminus c) \cup (c \ominus (a'\mathbb{Z} + b'))$$

□

5.5 A new solving step

Computing $\widetilde{S} = \theta_C(d_1, \ldots, d_n) \cap \rho_C(d_1, \ldots, d_n)$ by employing both propagation and elimination reduces the search space, because it allows the solver to quickly identify parts of the solutions. In fact, when the propagation of ρ_C is done, we propose an elimination step θ_C before splitting. Rather than performing arbitrary splits anywhere on a box, the elimination identifies parts of the box containing only solutions, and allows the solver to perform splits on the part of the search space that can not be discriminated as containing only solutions, nor as containing no solution. More precisely, elimination makes the split happen exactly at the frontier of the constraint.

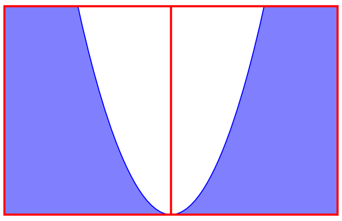


Figure 5.7: Splitting of a box

Figures 5.7 and 5.8 compare the results of the split of a box and the elimination followed by a split. We can see that the splitting frontier in Fig. 5.7 does not take into account the constraint while, in Fig. 5.8, the boxes containing only solutions are kept (boxes on both sides of the parabola). Then, the splitting operator splits the remaining box in two.

Where the Algorithm 6 gives the pseudo-code associated with our solving method with the new elimination step.

The difference with the default algorithm is in line 18–23. This algorithm processes elements that do not satisfy at least one constraint. The function `complement` computes $e_{non-cons}$, an over-approximation

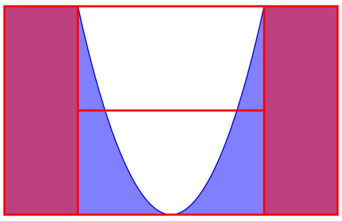


Figure 5.8: Difference, then split of a box

Algorithm 5 Solving without / with elimination

```
 1: function SOLVE($\mathcal{D}$, $C$, $r$, elim)    ▷ $\mathcal{D}$: domains, $C$: constraints, $r$: real, set elim to
 2:                                                    false for classic solving, true for elimination
 3:     sols := $\emptyset$                           ▷ sound solutions
 4:     undet := $\emptyset$                          ▷ indeterminate solutions
 5:     explore := $\emptyset$                        ▷ boxes to explore
 6:     $e$ := init($\mathcal{D}$)                    ▷ initialization
 7:     push $e$ in explore
 8:     while explore $\neq \emptyset$ do
 9:         $e$ := pop(explore)
10:         $e$ := filter($e$, $C$)
11:         if $e \neq \emptyset$ then
12:             if satisfies(e, $C$) then
13:                 sols := sols $\cup\, e$
14:             else
15:                 if $\tau(e) \leq r$ then
16:                     undet := undet $\cup\, e$
17:                 else
18:                     if !elim then
19:                         push $\oplus(e)$ in explore         ▷ Classic solving process
20:                     else
21:                         $(S, E)$ = elimination($e$, $C$)    ▷ Solving with elimination
22:                         sols := sols $\cup\, S$
23:                         push $\oplus(E)$ in explore
```

Algorithm 6 Elimination function

```
1: function ELIMINATION($e$, $c$)                 ▷ $e$: box, c: a single constraints
2:     $e_{non-cons} \leftarrow$ complement($e$, $C$)
3:     $e_{cons} \leftarrow e \ominus e_{non-cons}$
4:     $S \leftarrow \emptyset$
5:     for $e_i \in e_{cons}$ do
6:         $S \leftarrow S \cup e_i$
       return $(S, \oplus(e \cap e_{non-cons}))$
```

of the inconsistencies. Then, the difference operator is used to find the boxes containing only solutions. Finally, solving continues in the indeterminate search space $e \cap e_{non-cons}$ (instead of e).

Figure 5.9 shows the results obtained with our propagation/elimination/split loop on the CSP given previously, and gives for the same precision, much more satisfactory results: we require less elements to cover more space and in a comparable amount of time, showing that this technique deduces more relevant frontier than using a simple propagation/split loop.

5.5.1 For Several Constraints

When dealing with several constraints, conjunctively, the same method applies but can be quite imprecise. As the negation of a conjunction of constraint is a disjunction of constraint, using De Morgan's law, we explain in this section how computing symbolically the difference over each constraint can lead to more precise results. We thus propose this more precise procedure, that handles each constraint separately. It processes the constraints in turn applying each time the difference operator over the single constraint.

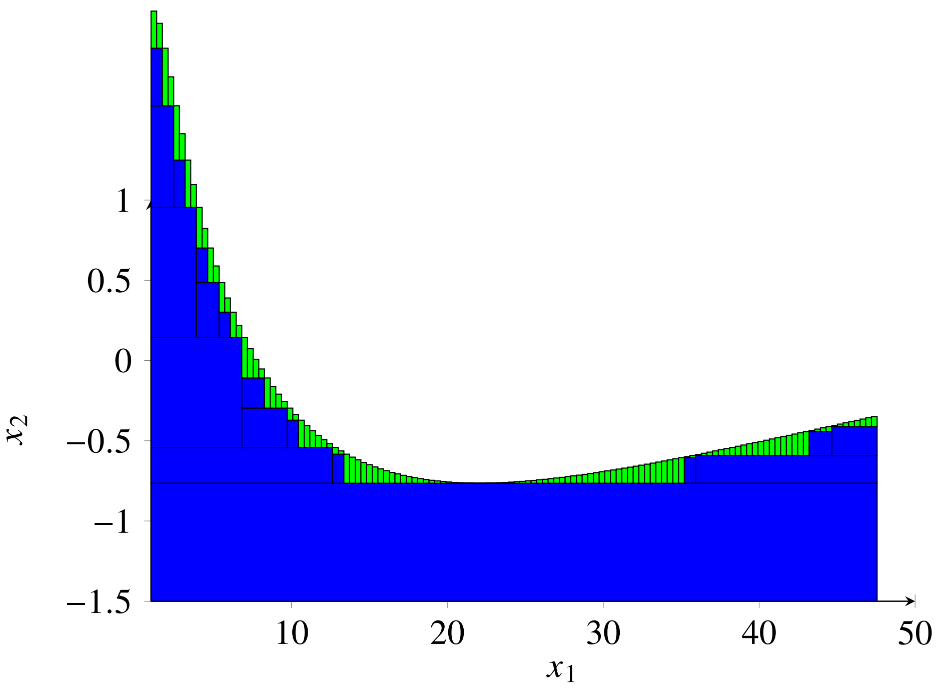


Figure 5.9: Solving with elimination : 125 inner boxes, 256 outer boxes. Inner boxes represent 98% of the coverage area. Computation time: 0.008s

Algorithm 7 Elimination for several constraints

```
1: function ELIMINATIONSET($e, C$)                ▷ $e$: box, $C$: a set of constraints
2:     Remove ← $\{e\}$
3:     Continue ← $\emptyset$
4:     while $C \neq \emptyset$ do
5:         $c$ ← pop($C$)
6:         NewRemove ← $\emptyset$
7:         for $r$ ∈Remove do
8:             $(S, x) \leftarrow elimination(r, c)$
9:             Continue← Continue ∪$x$
10:            for $s \in S$ do
11:                NewRemove ← NewRemove ∪$s$
12:        Remove ← NewRemove
    return (Remove, Continue)
```

From this constraint we separate the element that satisfy it from the rest, and verify if they satisfy the other constraint too. It gives us at the end of the traversal of the constraint store a list of abstract elements to remove from the original one, and a list of element where the search must continue.

Figure 5.10 illustrates the use of the elimination technique over an example with several constraints.

5.5.2 Difference for Disjunction Constraints

Thanks to this difference operator, we can now eliminate solutions from the search space but we can also improve precision and performances over the propagation of disjunctive constraints. Indeed, when the solver encounters a constraint of the form $c_1 \vee c_2$, it solves independently the problem with the constraint c_1 and with the constraint c_2 before joining and pushing the results into the solution list sol. With

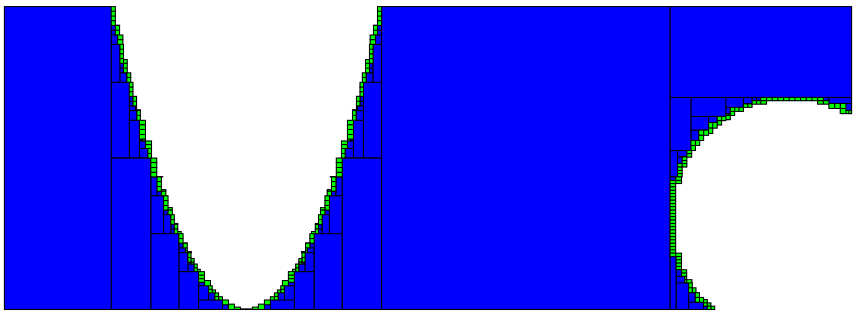


Figure 5.10: Resolution with elimination using simbolic difference over several constraints.

$a\rho(< \mathcal{X}, \mathcal{D} < \{c_1\})$ and $b = (< \mathcal{X}, \mathcal{D} < \{c_1\}$:

$$sol \leftarrow a \cup b$$

This of course can break our irredundancy property as $Sol(< \mathcal{X}, \mathcal{D} < \{c_1\} >)$ and $Sol(< \mathcal{X}, \mathcal{D} < \{c_2\} >)$ might intersect and even more so, their over-approximations. To solve issue we can now simply define an irredundant union of two abstract element a and b as:

$$sol \leftarrow a \cup (b \ominus a)$$

,

Which has the nice property of adding the intersection of a and b only once into the solution list.

5.6 Implementation and Benchmark

We now discuss implementation and performances aspects of this work.

5.6.1 Efficient Implementation

The general intuition of doing a propagation step on the negation of a constraint works correctly but can lead to an inefficient implementation. Satisfy test and pruning both require to work on the negation of the constraint and all of the three steps, satisfy test and pruning have to proceed to the same evaluation (Top-down) part of the HC4 algorithm. An idea that leads to a more efficient resolution is to factorize the evaluation part of the three steps.

5.6.2 Experiments

We have implemented our technique for boxes in the open-source solver AbSolute[3]. This solver is based on the method presented in [PMTB13b], where we integrated our elimination step. We rely on the abstract domain representation in AbSolute, which is based on constraints, to efficiently implement the constraint negation necessary for the elimination step. The unified constraint representation makes it possible to have an implementation of the difference operator that is lightweight and generic.

[3] https://github.com/mpelleau/AbSolute

5.6.3 Protocol

We tested our method on problems with continuous variables from the MinLPLib and the Coconut[4] benchmarks. For minimization problems, we first transform them into satisfaction problems, which can be handled by the solver. This transformation consists in adding an objective variable to the problem that will act as the value to minimize. Default bounds for unconstrained variables are set to -10^7 for the lower bound and 10^7 for the upper bound as our method requires the domains of the variables to be bounded. All of the runs are made with a time limit set to 300 seconds and no memory limit. Precision was fixed to 10^{-3} (i.e., the size limit where exploration stops), and branching depth was limited by 50. The solver was run on a Dell server with two 12-core Intel Xeon E5-2650 CPU at 2.20GHz, although only one core was used, and 128GB RAM.

We have tested the solving with the elimination step against the default solving method of the AbSolute solver over all of the problems that the solver's functionalities (types, constraint, arithmetic functions) are able to cover, that is 197 problems.

Table 5.3 shows the results obtained over problems of the minlp benchmark and table 5.1 presents the one obtained over the Coconut benchmark.

Table 5.1: Comparing solving with and without elimination step, Coconut benchmark

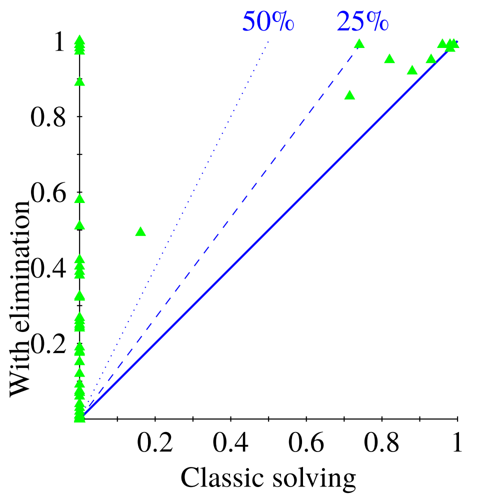
problem	$\|\mathcal{X}\|, \|C\|$	with elimination				without elimination			
		#I	#E	δ	t	#I	#E	δ	t
COCONUT problems									
abs1	1,2	2047	3072	0.99	**0.04**	4092	4096	0.99	0.06
aljazzaf	2,3	2309	19405	**0.58**	0.89	0	14319	0	**0.54**
allinitu	1,5	318	5066	**0.07**	3.26	0	5066	0	**2.50**
ampl	2,2	2	13930	**0.99**	0.66	0	13107	0.0	**0.46**
booth	1,2	90	45	0.12	**0.11**	0	45	0	0.13
cpr2ani10-10	10,10	0	14	0	0.09	0	14	0	0.09
dispatch	3,4	47	10642	**0.07**	3.39	0	15426	0	**2.42**
ex1411	2,5	1.78e6	2.59e6	0.98	**217.08**	1.95e6	3.74e6	0.98	237.89
ex1413	4,3	4884	34893	**0.25**	**0.52**	0	32698	0	0.78
ex_newton	2,5	638	950	**0.95**	**0.45**	729	892	0.93	0.57
griewank	1,2	19972	31868	**0.99**	**1.44**	29645	35105	0.98	2.30
h-s-f1	2,2	0	29	0	0.01	0	29	0	0.01
h73	3,4	19	1284	0.89	**0.06**	0	978	0	**0.05**
h76	3,4	24	174	**0.04**	0.05	0	82	0	0.05
hs23	6,2	825	2132	**0.99**	**0.43**	1315	1801	0.98	0.58
kear11	8,8	0	844	0	0.05	0	844	0	0.05
mickey	2,5	4315	12709	0.99	**2.40**	8372	9858	0.99	2.73
monfroy	3,4	0	745	0	0.08	0	745	0	**0.05**
nonlin1	2,3	1550	1978	**0.95**	**0.49**	2059	1772	0.82	0.69

[4]All informations about the problems can be found at http://www.gamsworld.org/minlp/minlplib/minlpstat.htm and http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html

nonlin2	3,2	4238	10560	**0.92**	**0.39**	8643	10692	0.88	0.42
tame1	2,3	29	3004	**0.51**	**0.21**	0	3177	0	0.31
zy2	3,3	6260	28147	**0.99**	1.00	13179	22499	0.74	**0.85**

Table 5.3: Comparing solving with and without elimination step, Minlp benchmark

		with elimination				without elimination			
problem	$\lvert X\rvert, \lvert C\rvert$	#I	#E	δ	t	#I	#E	δ	t
MINLP problems									
csched1a	23,29	0	8192	0	6.44	0	8192	0	**4.85**
deb10	130,183	0	0	0	0.01	0	0	0	0.01
dosemin2d	119,166	0	0	0	0.181	0	0	0	**0.177**
eg_all_s	26,8	1	54	**0.99**	2.99	0	16	0	**2.47**
eg_int_s	26,8	1	54	**1.00**	2.85	0	16	0	**2.35**
elf	39,55	0	3272	0	9.56	0	3276	0	**8.84**
ex1222	4,4	8	60927	**0.01**	1.39	0	61787	0	**0.97**
ex1223a	10,8	746	27097	**0.01**	**20.60**	0	48283	0	21.40
ex1223b	10,8	820	44084	**0.01**	40.22	0	500510	0	**29.41**
ex1252a	35,25	0	11	0	0.03	0	11	0	**0.02**
ex1264a	36,25	0	10544	0	8.80	0	10544	0	**6.26**
ex1266	96,181	0	20340	0	78.44	0	20340	0	**75.08**
gbd	5,5	576	31829	**0.19**	**1.27**	0	22927	0	0.93
prob03	2,3	0	625979	0	10.81	0	581831	0	**6.77**
procsel	8,11	0	2603850	0	289.98	0	2603850	0	**271.85**
qapw	256,451	0	0	0	**1.201**	0	0	0	1.233
sep1	32,30	0	26615	0	20.05	0	26615	0	**19.24**
st_e13	4,3	378	3102	**0.02**	0.05	0	18	0	**0.50**
st_miqp2	4,5	1352	38104	0.38	2.29	0	4564	0	0.31
st_miqp3	2,3	27	1117	**0.24**	0.03	0	1051	0	**0.02**
st_miqp4	5,7	3	21921	**0.01**	4.46	0	39152	0	**3.43**
st_miqp5	14,8	187	2080	**0.01**	4.71	0	6324	0	**2.38**
st_test1	2,6	1559	244337	**0.03**	18.70	0	231494	0	**15.97**
st_test2	3,7	4521	81280	**0.06**	**46.85**	0	338122	0	32.60
st_test4	6,7	116	33995	**0.01**	3.13	0	22076	0	**2.29**
st_test5	12,11	22	29520	**0.01**	7.82	0	11167	0.00	**7.55**
synthes1	7,7	97	33747	**0.01**	**4.18**	0	1285	0	5.16
tls2	25,38	0	18030	0	27.46	0	18030	0	**26.81**
windfac	14,15	0	19561	0	6.66	0	19561	0	**6.51**



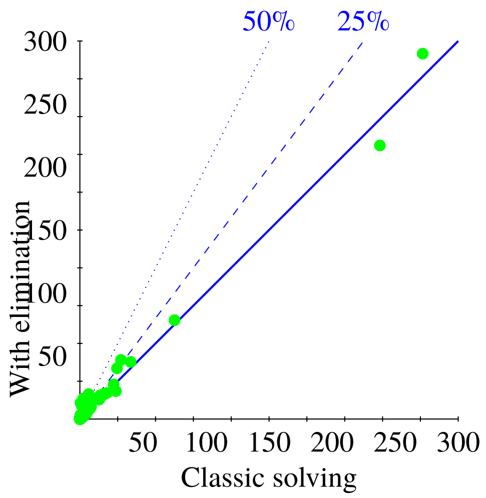


Figure 5.11: Comparison between the classic solving method and our method. On the left, comparison of the ratio, a mark above the bisector (in plain blue) means that our method is better than the classic solving. On the right, comparison of the computation time, a mark above the bisector (in plain blue) means that our method is slower than the classic solving.

5.6.4 Analysis

These runs highlight one very crucial feature of our method: it is able to quickly find boxes that contain only solutions of problems where the default solving method fails to do so (problems *aljazzaf*, *allintu*, *ex1222*, *gbd*, ...): on the whole benchmark, for almost 30% of the problems (58 out of 197), solving with the elimination step exhibited at least one solution while the default solving method did not succeed to do so. This comes for no time loss in average: on the whole benchmark, solving with elimination was slightly slower than without (1157 minutes against 1032 minutes). In fact, 39% of the problems (39 out of 197) were solved faster with the elimination than without (problems *ex1411*, *mickey*, *ex1223a*, *synthes1*...). This illustrates the fact that results of the solver are more precise: elimination avoids unnecessary splits, better identify the constraints frontiers, and compute within the same process inner and outer approximation for no (or little) overhead. A better analysis of the results shows that the default solving method spends time splitting variables with large ranges, while elimination focuses on the shape of the constraints to locates areas than can be directly removed from the search space and added to the solution set.

Figure 5.11 summarizes the results obtained with our method compared to the classic solving, regarding computation times and inner volume ratio.

Another conclusion of the analysis of this benchmark is about the solution coverage. The experiments show that the coverage of the solution space is significantly more accurate with the elimination step. On all of the runs, our method always find a greater or equal inner volume than the one found by the default method. Moreover, it also reduces the number of elements involved in the partition in the same time, which means that the inner approximation is achieved with less, bigger elements. This is shown by examples *chi* and *mickey* where both methods achieve a 0.99 ratio of inner volume, only with elimination, we need half the elements required by the default solving method to do so. On the whole benchmark, on average, we need 40 times less elements to cover the same inner volume with elimination. This property may become very handy as it allows a better re-usability of the results since we need to treat fewer elements to cover the solution space. The δ columns indicates the part of the returned elements that corresponds to an inner approximation, i.e. contains only solutions. This ratio is always greater with the elimination step. On the whole benchmark, the average ratio is of 0.49 of inner volume for the elimination while it is of 0.27 without. This confirms that the elimination step allows the solving process to target more efficiently the parts of the research space that contain only solution.

These good results confirm the intuition that cutting an element according to the constraints it does not satisfy can be more interesting than cutting it arbitrarily regardless of the constraints. Since solvers are often used as a pre-computation for other programs, reducing the size of their output (i.e., reducing the number of boxes required to represent a solution at a given precision) can be an important feature. Also, note that, by quickly identifying solutions and removing them from the search space, the elimination step makes it possible to carry out fewer propagation and exploration steps. Finally, it is interesting to notice that when all elimination steps fail to reduce the search-space, to resulting slow-down is constant, as at every step, we do two rounds of propagation instead of one. However the speed-up elimination may bring is depending on the problem and can be very significant.

5.7 Conclusion

5.7.1 Contributions

In this chapter, we have focused our work on improving the resolution technique based on abstract domains. The goal was to propose results that are both simpler to obtain and easier to reuse. In that context, we have developed a way of improving the quality of the results of a solver, while maintaining and on some examples improving the resolution times. Also, we believe that the covers computed with the elimination are more relevant to the end user and facilitate its work. Moreover, this technique integrate well with the reduced product and the mixed discrete-continuous representation we have define in chapters 4 and 3 as it is sufficient to lift the corresponding operator to these representations to have it available with these domains. Our implementation in the AbSolute solver shows good results using this technique.

5.7.2 Perspectives

Further work regarding the use of elimination include adapting it within global constraint propagators. Indeed, using the negation of a constraint does not always make sense when dealing with global constraints. Also, the design of activation heuristics is worth considering. For example, elimination proves to be very efficient when the solution ratio of the search-space is high. Such heuristics can be to activate it over inequalities constraint in priority, or over constraints known do reduce sparsely the solution space. In an Abstract Interpretation point of view, more work can be needed to lift this technique to other abstract domains such as ellipsoids or zonotopes which are not closed under intersection which may result in a different way of implementing the difference operator, probably in a more symbolic way.

Chapter 6

Constraint Aware Exploration Strategy

Abstract

In the previous chapter we have studied a way of improving the quality of the cover of the solution set, based on propagation. In this chapter, we continue with the same goal but with a different approach, focusing on exploration. We introduce a technique called *pizza splitting* which splits an element in several relevant parts, taking into account a constraint. Pizza splitting is based on a heuristic that allows it to find a key *point* of the search space to guide the split. From this point, it defines interesting splitting frontiers for the abstract elements. It reinforces the filtering potential of both propagation and elimination as the resulting split is likely to produce elements that can be discarded immediately. Combined with propagation and elimination pizza splitting allows the solver to focus on the frontiers of the constraints, which is the core difficult part of the problem and to reduce the number of unnecessary splits. The work we present in this chapter is still in progress and have not been fully implemented yet.

Contents

6.1 Motivation and Related Works

Solvers efficiency highly depends on the choices made during the exploration phase. Hence, whether by regarding the type of problem one has to solve, or in a more general purpose fashion, the design of specific exploration strategies is very often necessary to improve the solving process. For example, in [Ref04], the author proposes a search strategy based on the concept of the impact of a variable. The impact measures the importance of a variable for the reduction of the search space, and uses learning from the observation of domain reduction to guide the search. Other strategies involve a counting-based search instead of an impact-based one, as in [ZP09] where some constraint-centered heuristics are proposed to guide the exploration of the search space toward areas that are likely to contain a high number of solutions. More recently, in [ZMRM17], the authors introduce new strategies dedicated to floating point problems, that take advantage of the properties of floating point domains (e.g., domain density), to select values that are likely to provoke a rounding error. Closer to our work, *Mind The Gaps* [BMR05] uses the idea from [Han92, Rat94] and uses partial consistencies to find interesting splitting points within the domain, according to the “gaps" in the search space: splitting the domains by taking into account such gaps can lead to a significant reduction of the search space.

In this chapter, we propose a strategy for continuous solving. Its goal is to avoid the flaws of a standard bisection which can be irrelevant in some cases as it does not take into account the solutions-space configuration, *i.e.* the constraints, but only the search-space configuration, *i.e.* the variables. This has the consequences of producing partitions of the solution space or even making the method fail to find interior solutions for a given precision. The strategy we define adopts a constraint-centered approach to find a key *point* on the frontier of the search space, to guide the split. Better targeting the constraint boundary may be of crucial importance for defining inductive invariants as in [MBR16], where the choice of the splitting frontier has a strong impact on chances of success of the solving method. Note that we focus on proposing a better cover of the solution space, with respect to the metrics defined in chapter 3, without having to suffer a too large time overhead, and the speeding-up of the solving process is not our main goal.

6.2 Intuition

Thanks to a combination of propagation and elimination, we can now exploit at maximum the filtering capabilities of the constraints, both regarding the consistent instanciations and inconsistent ones. However when both propagation and elimination fail to remove elements from the search space, that means that the abstract element that encompasses the solution space and the one including its complementary are mingled. In this case the resolution entirely relies on a good exploration choice that would allow the solving process to get out of this situation where propagators are inefficient. In this section, we develop a new exploration strategy, meant to be used in this specific scenario that helps making less arbitrary splits than the standard bisection strategies. It exploits the intuition that a good split is likely to produce elements that can be directly discarded as satisfying the constraint or not. It can be decomposed into two steps:

- a point selection phase, tunable with several strategies, which should be used to guide the split,
- a splitting operator, that splits with respect to a point, which we call *pizza split*.

6.2.1 Example

Let us illustrate the problem we want to address by considering a concrete example. This example involves an ellipse constraint that accepts the interior of the ellipse.

Example 19. *An example with 2 variables $(x, y) \in \mathbb{R}^2$ subject to 1 constraint.*

- $x1 \in [-5; 5]$
- $x2 \in [-5; 5]$
- $u = 0.5$
- $v = 1.5$
- $t = 45$

Constrained with:
$(1/9) * ((x-u) * cos(t) + (y-v) * sin(t))^2 + (1/4) * ((x-u) * sin(t) - (y-v) * cos(t))^2 \leq 1$

Resolution:

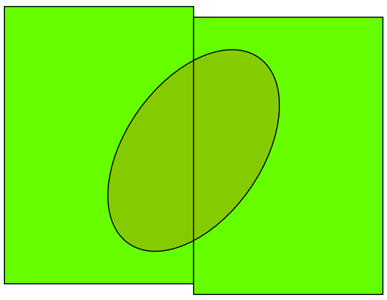
Abstract domain used	*Box*
Inner solutions	*38*
Inner volume	*14.28*
Outer solutions	*64*
Outer volume	*7.21*
Inner ratio	*6.45*
Solving time	*0.4s*
Precision	*0.001*

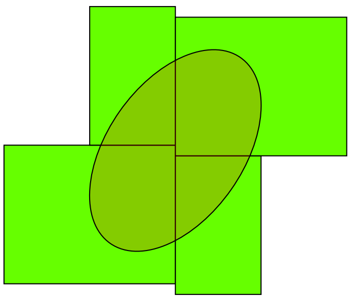
Figure 6.1 illustrates some of the first iterations of the solving process. Here the propagation reduces the current element to a smaller box that encompasses the ellipse, and the elimination is not useful as the inconsistent space is neither convex nor connected, which yields an important over-approximation: the smallest box that contains all the inconsistencies is the same that the one containing all the solutions. Due to this poor precision, the efficiency of the solving method will entirely depend on the the next exploration steps. Here, using a strategy such as the bisection of a variable is not very effective as the following propagation/elimination rounds will always remove a small portion of the search space. This can be explained by the fact that the elements produced by a split are unlikely to entirely satisfy the constraints or to be entirely inconsistent.

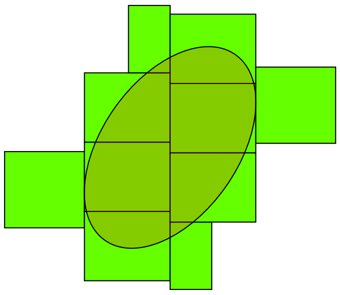
We are going to formalize the intuition that a better split could have yielded better results by producing elements that are likely to be in one of these two situations. Both propagation and elimination fail to reduce efficiently the search space as the solution space and the inconsistent one are both abstracted into the same element. Moreover, even after an exploration step, the obtained sub-problems find themselves in the same situation where propagation and elimination remain not very effective. By the end of the solving process, this has as an effect the production of a cover of the solution space made of a lot of small abstract elements. In the same spirit than in the previous chapter, we are going to build a technique that reduces the number of elements of the cover and avoid unnecessary computations.

6.3 Splitting According to a Constraint

Approximating with linear numerical abstract domain elements such as intervals or polyhedra a complex shape with non-linear constraints involved is a difficult task. Indeed, as these domains feature only linear constraints, it would require an infinite number of generators lying on the frontier of the constraint to approximate with no loss of precision a non-linear space. Our main intuition in this chapter is that the split operation can be much more efficient if it is guided: having a witness point of the boundary of a


(a) First iteration of the solving process


(b) After 2 iterations


(c) After 4 iterations

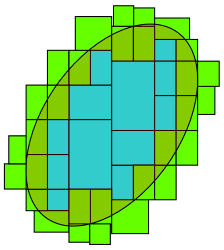

(d) After 16 iterations

Figure 6.1: Propagation/Elimination/Split loop over Example19

constraint can help defining relevant cutting frontiers for this constraint. Another way of understanding this idea is that if we find one point, on the boundary of the considered constraint, and enforce the resulting elements of the next split operation to have this point along their vertices or situated on one of their constraint, then by the end of the solving process, our abstract elements will have a significant part of their generators on the frontier of the constraint. We believe that the partition of the solution space obtained that way will be of a particular interest as it would result in less arbitrary splits, and thus less elements in our cover.

To define a relevant exploration of an abstract element with respect to a constraint, we propose to divide the problem into two parts: the definition of a heuristic *keypoint* that determines a key point in relation to the constraint, and the splitting of this element with respect to this key point using a new split operator called *pizza split* and noted ($\otimes$). Both of these part are meant to be used as illustrated by Algorithm 8.

Algorithm 8 Splitting of an element according to a constraint

```
function SPLIT(e,c)                    ▷ e: an abstract element          ▷ c: a constraint
    p ← keypoint(c)
    output(⊗(e, p))
```

We will firstly explain how to split an abstract element according to a point, and then discuss the different heuristics possible to find such a point. Finally we incorporate this technique to our method of resolution and illustrate its operation on some examples.

6.4 Splitting According to a Point

In this section we focus on how to split an abstract element, once an interesting point has been found, using this point. An interesting idea is that if the point is on the boundary of a constraint and it is also on the boundary, or on one of the vertices, of an abstract element, then there is a chance that the whole abstract element is on one side of the constraint or the other. In both cases, propagation or

elimination should be able to remove it from the search space. Following this idea, we now define a split operation that takes not only an abstract element but also an instance ($\mathcal{X} \to \mathcal{D}$), and splits the element in such a way that all of the resulting element have this instance as one of their vertices or situated on one of its constraints. This split operator should respect the same properties as in 21.

Definition 60 (Pizza split operator)**.** *A pizza split operator $\otimes : \mathcal{B} \times (\mathcal{X} \to \mathcal{D}) \to P(\mathcal{B})$ is a binary operator such that:*

$|\otimes(e, p)|$ *must be finite, ensuring the finite the width of the search tree,*

$\forall e_i \in \otimes(e, p), \tau_E(e_i) < \tau_E(e)$ *ensuring finite depth of the search tree (termination),*

$\cup \otimes (e, p) = e$ *enforcing that splitting does not lose solutions (completeness),*

and $\forall e_i \in \otimes(e, p), e_i = e \implies e$ *is a least element of E ensuring that the splitting operators does not lose nor create elements (completeness and soundness).*

From this definition, we are going to define a splitting strategy that splits an element in a more aggressive way than a standard bisection, and takes advantage of the point used.

6.4.1 Pizza Split Operator for Boxes

As boxes are Cartesian products of intervals, we can decompose the definition of the split operator into a split over intervals and lift this operator to the Cartesian product.

Definition 61. *Given a real interval $i = (l, b_l), (u, b_u)$ and a value $s \in itv$, with $ls < u$ the pizza splitting of i according to s is:*

$$\otimes_{itv}(i, s) = i_1, i_2$$

with:

$$i_1 = ((l, b_l), (s, true)) \tag{6.1}$$
$$i_2 = ((s, false), (u, b_u)) \tag{6.2}$$

Here, we voluntarily impose that the point is different from the bounds of the interval, so that the cutting operator is effective. Moreover, as we will see later, the way in which we determine these points will ensure that they can not be on the edge of an interval. This split is complete and correct as it produces a perfect cover of the original interval. Moreover both of the resulting intervals have the given point as one of their endpoints. Also, note that the split operator given in 4.3.4 can be seen as a particular of this one, where s is equal to the middle point of the interval $\frac{l+u}{2}$.

From this split, we can define the split over the boxes abstract domain.

Definition 62. *Given a real box $B = \{i_1 \times i_2 \times \cdots \times i_n\}$ and an instance $s = \{x_1 \to v_1, x_2 \to v_2, \ldots, x_n \to v_n\}$, with $s \in \gamma(B)$, the pizza splitting of B according to s is:*

$$\otimes_{box}(B, s) = B_n$$

with:

$$\begin{aligned} B_n &= \{i \times b | b \in P(B_{n-1}), i \in \otimes_{itv}(i_n, s(v_n))\} \\ B_1 &= \{i \in \otimes_{itv}(i_n, s(v_1))\} \end{aligned}$$

This operator corresponds to all the boxes we obtain by repeating the split along a variable, and splitting the resulting elements along the next variables, until we have split all of the variables. Figure 6.2 illustrates the results of this split operator.

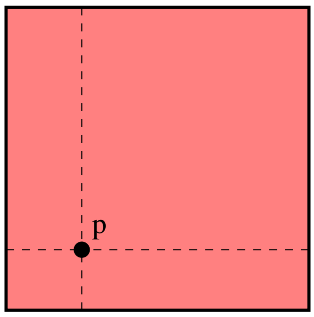


Figure 6.2: Pizza split operator for the interval abstract domain

It is interesting to notice that this split creates more elements than the standard bisection as it created 2^n boxes against 2, where n is the number of variables. However, it is based on the idea that the elements are chosen in a better way, and will be easy to discard from the search space during the next propagation rounds and elimination rounds. Also, we can split only the variables appearing in the constraint involved, to avoid the creation of a too large number of abstract elements.

Proposition 11. *Pizza split The Pizza split operator for the box abstract domain is correct, complete and irredundant.*

Proof. As the split over interval is correct complete and irredundant, by definition of the Cartesian product, $\otimes$ is trivially complete, correct and irredundant. □

Pizza Split Operator for Polyhedra

Splitting a polyhedron according to a point is more difficult: in two dimensions, a solution is to find a triangulation of the polyhedron such as all of the resulting triangles have the given point as one as their vertices as illustrated in Figure 6.3.

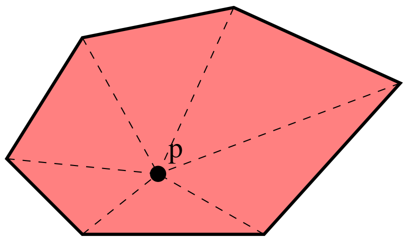


Figure 6.3: Triangulation of a two dimensions polyhedron, according to the point p

In higher dimensions, one possibility is to divide a polyhedron into a set of simplices such that all of the simplices have the given point as one of their generators, but this is not trivial. For example, it is NP-complete to compute the smallest number of tetrahedra needed to triangulate a convex 3-polytope [BDLRG04]. Nevertheless, we can still look at different solutions that are easier to calculate. For instance, we can use the same strategy as for boxes and split along the variables of the polyhedron as shown in Figure 6.4.

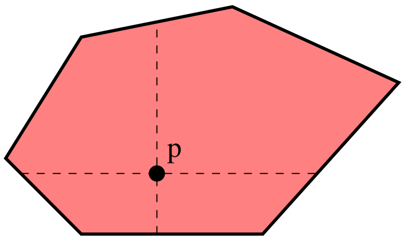


Figure 6.4: Pizza split operator for the polyhedra abstract domain

Another strategy, closer to the splitting of polyhedra introduced in chapter 3 is to build splitting frontiers from the generators of the polyhedron. We can for all the generators of the polyhedron compute a line l that goes through this generator and the splitting point. We can then build two polyhedron with the constraint $l > 0$ in one and the constraint $l \leq 0$ in the other. We can repeat this operation with the obtained sub-elements for the rest of the generators.

Also, in order to not over-split we can refine this strategy by beginning with the closest generators, and stopping as soon we have handled a certain number of the generators, or as soon as the generators are further than a given threshold. This reflects the idea that the closer a generator is to the splitting point, the more the elements resulting from the cut having this generator as one of theirs will have a chance of being entirely on one side or the other of the constraint. When the generators are more distant from the cutting point, the resulting elements are bigger, and thus less likely to be entirely on one side of the constraint. The result of this method is illustrated in Figure 6.5.

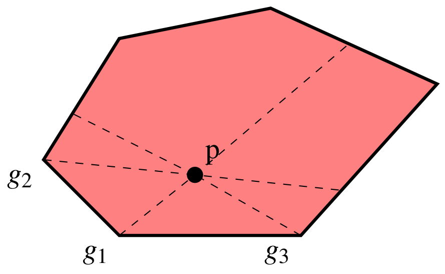


Figure 6.5: Pizza split for the polyhedra according to point p, from the three closest generators g_1,g_2 and g_3

6.4.2 Pizza Split Operator for the Reduced Product

To have our new exploration strategy enabled within our reduced product abstract domains, we define the corresponding pizza split operator. Its definition within our specialized reduced product is very natural as our product uses a dispatch function to attribute a constraint to one of its components. Hence, we only have to select one abstract domain according to the constraint, and perform the pizza splitting with it. However the definition is now made at the constraint level split, and not anymore at the point level, as the product uses the constraint to define which of its component will be assigned the operation.

Definition 63. *Let c be a constraint, and $D^\sharp_{A\times B}$ a product of domains $D^\sharp_A$ and $D^\sharp_B$, and δ its dispatch function. The split operator* $\text{split}_{A\times B}$ *according to a constraint is defined as follows:*

$$\text{split}_{A\times B}((a,b),c) = \begin{cases} \{(x,b) | x \in \text{split}_A(a,c)\} & \textit{if } \delta(c) = true \\ \{(a,y) | y \in \text{split}_B(b,c)\} & \textit{if } \delta(c) = false \end{cases}$$

6.5 Finding a Point on the Boundary of a Constraint

Now that we have a splitting operator able to divide abstract elements according to a point, we have to define a way to find an interesting point for the split operation. In this section we focus on finding a point on the boundary of a single constraint of a numerical CSP. We recall that a point is an instance which is a total mapping from variables to real values.

Definition 64. *Given a set of variables $\mathcal{X}$ and their domain $\mathcal{D}$, the boundary of an inequality constraint c of the form $e_1 \bowtie e_2$, with $\bowtie \in \{<,>,\leq,\geq\}$, is the set of points p such that:*

$$\{p \in Sol(<\mathcal{X},\mathcal{D},\{c'\}>),\ with\ c' = e_1 = e_2\}$$

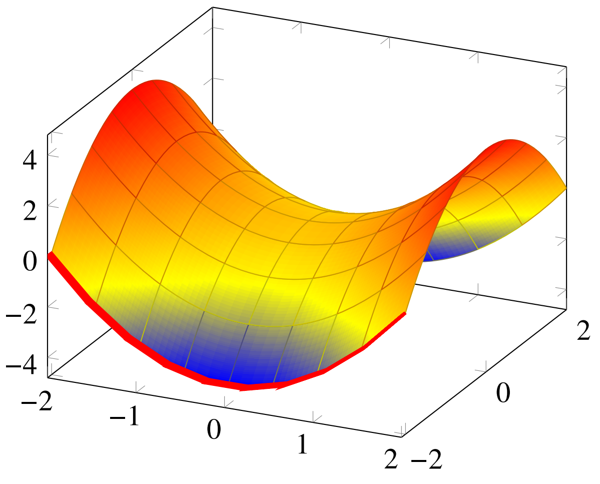
If the given constraint is an equality of the form $e_1 = e_2$, in this case we will look for only one arbitrary point that satisfies it. If it is an inequality $e_1 \leq e_2$ or $e_1 < e_2$, then we reduce the problem to the first case by looking at the constraint $e_1 = e_2$. In both cases, the problem to be solved is simpler than the original problem, because we are now searching for a single solution of an equality constraint rather than all of the solutions of an arbitrary system of constraints.

6.5.1 In Turn Instanciation

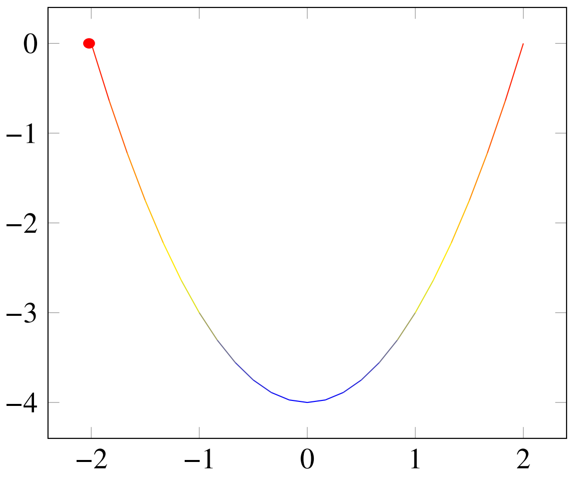
The first idea for finding a point lying on the boundary of a constraint $e_1 = e_2$ is simply to instantiate each variable in turn to a given value of its domains, and verify if the instance we obtain satisfies the constraint. Of course doing a propagation round between the successive instanciations makes it possible to detect early failures and guide the next instanciations. Moreover, we can reduce the search-space by considering only the variables appearing in the considered constraint as there is no need to instantiate the other variables. This method has been studied a lot in Constraint Programming and following this idea, we can use various heuristics to determine the order of instanciation of the variables. For instance, the one proposed in [Bré79] chooses the variable having the smallest domain (dom) and appearing in the largest number of constraints deg. In others words, the chosen variable is the one that maximizes $dom + deg$. Another related way of choosing a variable, presented in [BR96] is to choose the variable maximizing the $(dom)/deg$ ratio. Once the variable to be instantiated chosen, we have to choose by which value to instantiate it. Here too, a lot of different strategies have been developed, choosing the value maximizing the number of possible solutions [DMP89, KDG04] or the sum of the domains size[FD95] or even a simple bisection.

Median. In our case, a heuristic that is not necessarily guaranteed to find a point is sufficient. In case it fails, we can still use the standard bisection method. Thereby, a naive heuristic to apply is the following: we first instantiate the discrete variables using the *smallest-first* heuristic to the median value of their domains and then the continuous one also to their median value using the *largest-first* heuristic. We also perform a propagation round between each instanciation to detect early failures and narrow the possibilities for the next variable instanciations. This heuristic is based on the idea that splitting according to a central point, then we avoid the creation of too small elements that, when if removed from the search space, will not significantly reduce its size.

Exploiting consistency. Another idea is, as we perform filtering rounds during the instanciation process, to exploit the characteristics of the propagators being used. For example, with the interval abstract domain, we use the HC4 algorithm which computes the smallest enclosing box of a space defined by a constraint. This guarantees[1] that the obtained hull contains solution points on its boundaries: this is the *hull-consistency* property. Hence an interesting heuristic is to choose for each variable one of its bounds, perform a propagation round and repeat the process for the next variables. Once an instance is found using this method, we still have to check that it indeed satisfies the constraint as the HC4 algorithm compute a smallest *floating-point* box that encompasses a *real* space. If it is the case, the heuristic succeeded.



(a) Fixing variable y to -2



(b) Fixing variable x to -2

Figure 6.6: Point selection strategy

Figure 6.6 illustrate this technique over the constraint below, where in each sub-figure the highlighted part in red is the part remaining of the solution space after fixing a variable:

$$z = x^2 - y^2$$

[1]disregarding possible rounding errors

With the corresponding domains : $x \in [-2;2], y \in [-2;2] z \in [-4;4]$. After a first round of propagation, the variable y is fixed to its minimum value -2, which gives a constraint with only two variables left, x and z:

$$z = x^2 - 4$$

After a new round of propagation, we are able to narrow the domain for z to $[-2;0]$. We now can fix the variable x to its minimum value -2. A last round of propagation fixes the remaining variable z to 0. The heuristic succeeded and find the point: $(-2,-2,0)$.

Tuning the Heuristic

One flaw of this point selection strategy is that it chooses a point that is on the boundary of the initial abstract element. This can be problematic because the sub-elements resulting from the cut can also choose this same point during their exploration step. In this case the pizza split will be ineffective. To avoid this problem, we propose to narrow the selection of a point to an interior region of the abstract element. This is done by a shrinking operation that computes an homothetic transformation according to the centroid of the abstract element as illustrated in Figure 6.7.

Definition 65. *Given an abstract element defined by a set of generators G, its shrunk associated element is given by the set of generators G' such that:*

$$G' = \{B + \lambda\overrightarrow{Bg} | g \in G\}$$

Where B is the centroid of G, $\overrightarrow{Bg}$ the vector going from B to g and λ a real value in $]0;1[$

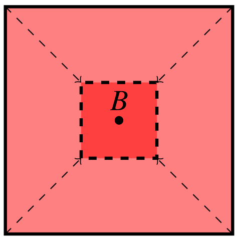
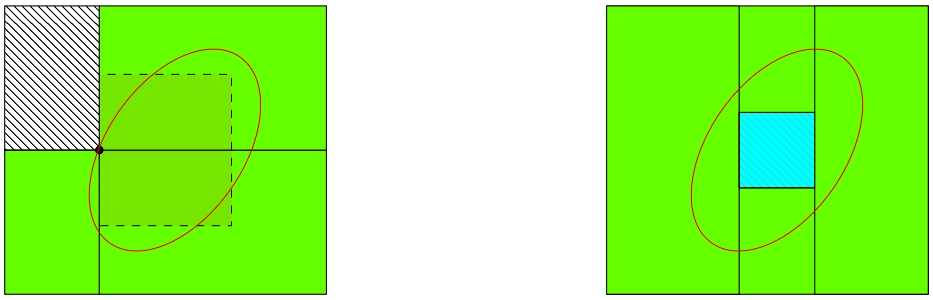


Figure 6.7: Shrinking operation of a box

Here, it is important to select a ratio λ different from zero otherwise the selected point is always the centroid of the abstract element, and different from one otherwise the shrunk element is the element itself. Moreover, the interest of this shrinking operation is twofold: it makes it possible to avoid the selection of a point on the boundary of the abstract element, but can also make it possible to detect that the boundary of the constraint considered is inconsistent with the shrunk element. Indeed, as we perform propagation steps during the point selection, we can perform one before fixing a variable. From there, two cases are possible:

- either the shrunk element is consistent with the boundary of the constraint, in which case we can proceed to the point selection,

- either it is inconsistent in which case we have still learned some information about the problem: the shrunk element either entirely satisfies the original constraint, or is entirely inconsistent with it. We can in this case use it to define a splitting operation using the difference operator introduced in the previous chapter.



(a) e intersects the frontier of c (b) e does not intersect the frontier of c

Figure 6.8: Different possible splittings according to a constraint c, and a shrunk element e

Figure 6.8 illustrates these two different possibilities and the resulting splitting operations, with our running example problem. In both cases, we have managed to build a partition of the original abstract element, where one member of the partition (the hatched one) can be discarded from the search space. In the first case it is inconsistent, and in the second, it satisfies the constraint.

Also, note that randomizing the selection of the ratio λ can be interesting to make the behaviour of the heuristic non-deterministic. This allows to embed it in a *k-retry* procedure: when the heuristic fails to find a point, it is allowed to retry k times, by taking a bigger value for λ at each try, before failing.

6.5.2 Gradient Descent

Another way to find a point on the boundary of a constraint, is to reduce the problem to an optimization problem. For example we can use a gradient algorithm to perform this task. This is a differentiable optimization algorithm and is therefore intended to minimize a differentiable real function defined on an Euclidean space. The algorithm is iterative and therefore proceeds by successive improvements. At a current point, a displacement is made in the direction opposite to the gradient, so as to decrease the function. The key here, is given an inequality $e_1 < e_2$, finding a minimization problem whose solution is a solution for $e_1 = e_2$. This is achieved by minimizing $(e_1 - e_2)^2$.

Indeed, as $(e_1 - e_2)^2$ is always positive, the minimum possible value it can take is 0, and when it is the case, $e_1 - e_2 = 0$. If the minimum value we obtain is equal to 0, then the heuristic succeeded and the instance for which $(e_1 - e_2)^2 = 0$ is on the boundary of the constraint. Otherwise, we can choose to split the element using this point, if the minimum of the expression $(e_1 - e_2)^2$ is close enough to 0. Indeed, with floating calculations with rounding, it may be very unlikely (or even impossible in some cases) to find exactly one instance for which the expression $(e_1 - e_2)^2$ is equal to zero.

6.6 Implementation and Preliminary Results

We have implemented the *pizza split* technique inside AbSolute Even though this implementation work is still at an experimental stage, we have been able to test the performances of the *pizza split* on some

Table 6.1: Comparing the pizza split and the standard bisection, with respect to computation time and inner ratio.

#var	inner ratio, Standard	inner ratio, Pizza	time, Standard	time, Pizza
2	0.280	**0.648**	0.007	0.007
2	0.882	**0.907**	**0.011**	0.012
2	0.280	**0.986**	**0.032**	0.052
2	0	**0.001**	**0.371**	0.564
2	0	**0.013**	**1.932**	2.752
3	0	**0.019**	**6.817**	9.899
3	0.579	**0.795**	**0.787**	0.654
3	0.280	**0.643**	**0.856**	1.227
4	0.891	**0.937**	**1.401**	2.644
5	0	0	**20.401**	30.982

examples and the first results obtained are encouraging. Table 6.1 summarizes some of the results obtained with as the propagation based point selection heuristic. Constraints and domains have been selected from real problems of the MinLp benchmark. The first column indicates the number of variables involved in the considered constraint. The second and third column compare the inner ratios obtained with the standard split and with our new technique, and the last two column show the corresponding resolution times in seconds.

From the analysis of the result, we notice that our new split strategy tends to be slower than the standard one but the splitting it defines yield better inner-approximations of the solution space. The fact that the inner ratio is always better with the *pizza split* technique confirms that it helps discarding areas immediately where the standard bisection would create unsure elements, and in particular we can notice that with the *pizza split* enabled, our resolution method is able to find solution for 3 of 1O problems where the standard bisection fails to do so.

6.7 Conclusion

6.7.1 Contributions

In this chapter, we have continued the objective fixed by chapter 5 to improve the quality of the results of a solver. Our main idea was to guide the split, to define relevant splitting frontiers, when propagation and elimination were ineffective. We have proposed a new split operator, that splits according to a point, along with some point selection strategy to guide the exploration step and build a more interesting partition of our abstract elements.

6.7.2 Future Work

Our perspective of continuation include the experimentation of the pizza split operator on realistic benchmarks, with the different point selection heuristics we have presented. Adapting this technique to several constraints will also be a necessary step to make its use effective. Also, we believe that the development of activation heuristics will be required to make this split operator efficient. These could be its activation on non-linear constraints, or on convex shapes for which this operator seems to give the best results according to our preliminary experimentation. Also, the general idea of finding one solution instance of a problem to guide the building of the whole solution set can be interesting and adapted to other solving techniques.

Chapter 7

Conclusion

In this thesis, the aim was to propose a tight collaboration between the techniques of Abstract Interpretation and Constraint Programming within a unified method of resolution of constraint satisfaction problems. This work addresses the problem of the design of a Constraint solver based on abstract domains, in a generic and modular way. It uses both the theoretical and practical basis of Abstract Interpretation combined with some standard techniques of Constraint Programming, and some ones, to bypass the restriction of more standard constraint solvers, such as the dedication to a certain type of constraints or variables. Our effort also consisted in the design of a robust method, aware of floating-point errors with a sound handling of rounding errors. This method is implemented within the AbSolute constraint solver and applied on several examples. Experiments prove that our methods improve the efficiency and provide more relevant answers. In this chapter we briefly recall the contributions of our work and discuss the future works.

7.1 Contributions

We have defined in the Chapter 2, **Preliminaries**, the theoretical basis necessary to the understanding of our work. We have then situated our task within the state of the art of both Abstract Interpretation and Constraint Programming and in particular, in the continuity of the work of *Pelleau & al.*, in [Pel12, Pel15, PMTB13a]. We have during this thesis continued this work both from a theoretical and practical point of view.

In the two next chapters, **Abstract domains and domain products for Constraint Programming**, and **Discrete and Continuous abstractions for constraint solving**, we worked on the generalization of abstract solver concepts. The benefits of this generalization are threefold: it allows us to handle more efficiently different kinds of constraints, handle more problems, and define a generic way of augmenting a solvers capabilities. In particular, the third chapter replaces the standard notion of domains (in the CP sense) by more general representations imported from Abstract Interpretation. This is done in order to exploit the relational domains of Abstract Interpretation in a framework of constraint resolution. This allows us a more powerful mechanism of propagation, which reduces the resolution time. We use these constructions to define a standard way of augmenting the kind of constraints a solver is able to solve, using an abstract domain combinator: the reduced product, adapted to Constraint Programming purposes. The fourth chapter continues with the idea of generalization by focusing on mixing different types of variables. We proposed an abstraction that fits both discrete or continuous variables, based on a product of both real and discrete intervals, and the congruence abstract domain. This representation comes along with the corresponding propagators, splits and size operators.

In the two following chapters, **Propagation with Elimination** and **Constraint aware Exploration**, we have focused our attention on the development of techniques to obtain more relevant results from a point of view of their reusability. In particular, the cardinality of the partition of the solution space is minimized and its inner ratio is maximized. Also, the results we obtain suit both complete and correct resolution techniques and provide a guarantee of non-redundancy: no solution is covered by more than one abstract element. Chapter four explicits a new technique called *elimination*, whose purpose is to improve the quality of the results of a solver by reducing the search space by removing from it, as soon as possible, parts that are solutions. This reduction of the search space is able to significantly reduce the computation time on some example. It also leads to better results according to the metrics we have defined for comparing different covers of a solution space, moreover, it integrates well using the abstractions we have defined in the previous chapters. The sixth chapter presented ongoing ideas that follow the same purposes. We define in it a new exploration strategy, aware of the constraints of a problem and called *Pizza split*, meant to avoid the flaws of a standard bisection. Its goal is to find relevant cutting points within an abstract element, and split it accordingly, using a dedicated split operator.

Our work has been concretized under the form of an implementation in the AbSolute solver. The results we have obtained with it are very promising and validate on a practical level the techniques we have built. Also, it is interesting to notice that our implementation differs from most of the modern implementations of constraints solvers by its functional style. The use of immutable data structures, used in recursive procedures, and duplicated during branching in the search space, has allowed us to completely avoid the implementation of backtracking techniques, which usually represent an important part of the development of constraint solvers. This choice of paradigm is fundamental with respect to performance, and our good results show that a functional style is as well adapted to this problematic, although our implementation has always privileged genericity and modularity over performance.

7.2 Perspectives

7.2.1 Short Term

The promising results obtained with the AbSolute constraint solver open the way to the development of hybrid solvers, at the frontier of Constraint Programming and Abstract Interpretation able to naturally handle different representations. In a short term perspective, we wish to improve our solving method by adapting and integrating advanced methods from the Constraint Programming literature, in particular specialized propagators for global constraints, under the form of a specialized reduced product. The AbSolute solver is built on abstractions in a modular way, so that new methods and already existing ones can be combined together without difficulty. This opens many possibilities for mixing abstract domains, and new heuristics will be developed for choosing the appropriate abstract domain depending on the nature of the constraints involved in a problem. Ultimately, each problem could be automatically solved in the abstract domains which best fits it, thus avoiding redundant computations and enjoying a better precision. Also, now that we are able to handle efficiently both discrete and continuous variables, we can incorporate standard heuristics over the discrete representations inside our solver, and this also opens the possibility of new global constraints inside the AbSolute constraint solver, dedicated to discrete problems.

Further work regarding the use of elimination include adapting it within non-arithmetic constraints. Indeed, using the negation of a constraint is not straightforward when dealing with global constraints, and when it is the case, the design of a new propagator may be necessary. Also, the design of activation heuristics is worth considering. For example, elimination proves to be very efficient when the solution ratio of the search-space is high, and symmetrically ineffective when a problem is unsatisfiable. Such heuristics can be used to activate it over inequality constraints in priority, or over constraints known to reduce sparsely the solution space. From an Abstract Interpretation point of view, more work can is needed to lift the splitting and difference techniques to other abstract domains such as ellipsoids or zonotopes which are not closed under intersection, and it is not trivial to find a cover of such elements, not to say a partition. This may result in a different way of implementing the split operator and difference operator, potentially in a more symbolic way. Further work regarding the use of the pizza split inside solvers includes firstly a rigorous benchmarking of its capabilities against other exploration strategies, and in a second time, a study of its adaptation to other constraints. Of course the design of activation heuristics may be a necessary step to improve the performance of this split operator, for example, it may work best when a continuity hypothesis over the shape of the constraints is known, or when the key points are easy to determine.

7.2.2 Long Term

The mixing of discrete and continuous abstractions not only improves the efficiency of the our solving method but also broadens the spectrum of possible applications. A lot of real world problematics may enjoy the use of a mixed technique able to handle both discrete and continuous representations transparently: from computer-aided design, to robotics, the continuous nature of problems appear as soon as the problems take into account physical constraints (volumes, weights ...) and the discrete part arises from constraint such as the number of available resources (number of blades in a fan or the number of container for layout problems). From a broader point of view, our work on reduced products could also provide a basis for solver co-operation, something that is widely used in Constraint Programming and Operation Research, but in practice in an *ad-hoc* fashion. For example, the reduced product of intervals and polyhedra can be seen as a cooperation between a continuous constraint solver and an LP-solver (linear programming) and, in the same way, one could imagine a reduced product involving CP techniques and SAT-solvers.

Also, our efforts have mainly focused on solving constraint satisfaction problems, and it would be interesting to see to what extent the techniques we developed are reusable in the context of optimization problems. Other work ideas may relate to the development of constraint propagators for user-defined constraints. Indeed, as in Abstract Interpretation, the analyzers are able to handle the definition of new functions, during a programs analysis, we could exploit this idea to have it enabled for user-defined constraints.

Finally, this thesis had as main concern the application of Abstract Interpretation techniques for constraint solving and a natural question is to what extent can program verification take advantage of Constraint Programming methods? Constraint Programming provides efficient techniques to improve the precision of an abstraction, using expensive computations to reach an arbitrary given precision. Abstract Interpretation does not provide such mechanisms to improve on-the-fly the precision of the abstract domain being used. Incorporating some Constraint Programming mechanisms (splits, search tree) in abstract domains would enable, in the presence of false alarms, the automatic refinement of the abstraction being used and could help discard false alarms.

Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[AGM15] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny-cp: a sequential cp portfolio solver. pages 1861–1867, 04 2015.

[AHP$^+$18] Christian Artigues, Emmanuel Hébrard, Yannick Pencolé, Andreas Schutt, and Peter J Stuckey. A Study of Evacuation Planning for Wildfires. In *The Seventeenth International Workshop on Constraint Modelling and Reformulation (ModRef 2018)*, page 17p., Lille, France, August 2018.

[AW07] Krzysztof R. Apt and Mark Wallace. *Constraint logic programming using Eclipse*. Cambridge University Press, 2007.

[BCC$^+$02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, October 2002.

[BCC$^+$03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[BCC$^+$12] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A simplex-based extension of fourier-motzkin for solving linear integer arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 67–81, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BDLRG04] Alexander Below, Jesús A. De Loera, and Jürgen Richter-Gebert. The complexity of finding small triangulations of convex 3-polytopes. *J. Algorithms*, 50(2):134–167, February 2004.

[BDM03] Michael R. Bussieck, Arne Stolbjerg Drud, and Alexander Meeraus. Minlplib - A collection of test models for mixed-integer nonlinear programming. *INFORMS Journal on Computing*, 15(1):114–119, 2003.

[Ben96] Frédéric Benhamou. Heterogeneous constraint solvings. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, pages 62–76, 1996.

[Ber10] Nicolas Berger. *Modélisation et résolution en programmation par contraintes de problèmes mixtes continu/discret de satisfaction de contraintes et d'optimisation. (Modeling and Solving Mixed Continuous/Discrete Constraint Satisfaction and Optimisation Problem).* PhD thesis, University of Nantes, France, 2010.

[Bes01] S. Bespamyatnikh. An efficient algorithm for the three-dimensional diameter problem. *Discrete & Computational Geometry*, 25(2):235–255, Mar 2001.

[BGG97] Frédéric Benhamou, Frédéric Goualard, and Laurent Granvilliers. Programming with the DecLIC Language. In *Proceedings of the 2nd International Workshop on Interval Constraints*, 1997.

[BGGP99] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revisiting hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming*, pages 230–244, 1999.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI'2004)*, pages 146–150. IOS Press, 2004.

[BMR05] Heikel Batnini, Claude Michel, and Michel Rueher. Mind the gaps: A new splitting strategy for consistency techniques. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2005.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, pages 128–141, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Bou09] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323 – 341, 2009. Proceedings of the 5th Workshop on Methods for Modalities (M4M5 2007).

[BR96] Christian Bessière and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*. Springer, 1996.

[Bré79] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, 1976.

[CC77] Patrick Cousot and Radhia 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*, pages 238–252, 1977.

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, pages 312–327, 2004.

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Proceedings of the 14th International Conference on Fondations of Software Science and Computation Structures (FoSSaCS)*, pages 456–472, 2011.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, pages 52–71, 1981.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[CIM13] Sylvain Conchon, Mohamed Iguernelala, and Alain Mebsout. A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo. working paper or preprint, 2013.

[CJ09] Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173:1079–1100, 2009.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, 1978. Universités : Université scientifique et médicale de Grenoble et Institut national polytechnique de Grenoble.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[Cum16] David Cummings. Embedded software under the courtroom microscope: A case study of the toyota unintended acceleration trial. *IEEE Technology and Society Magazine*, 35:76–84, 12 2016.

[DAC12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of gnu prolog. In *Theory and Practice of Logic Programming*, volume 12, pages 253–282. Cambridge University Press, 2012.

[DCM08] Mohammad Dib, Alexandre Caminada, and Hakim Mabed. Propagation de Contraintes et Listes Tabou pour le CSP. In Gilles Trombettoni, editor, *JFPC 2008- Quatrièmes Journées Francophones de Programmation par Contraintes*, pages 409–413, Nantes, France, June 2008. LINA - Université de Nantes - Ecole des Mines de Nantes.

[Dil90] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, pages 394–397, 1962.

[DMP89] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, 1989.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, pages 201–215, 1960.

[FD95] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the 14th International Joint Conference on Artificial intelligence (IJCAI'95)*, pages 572–578. Morgan Kaufmann Publishers Inc., 1995.

[Fer04] Jérôme Feret. Static analysis of digital filters. In Springer, editor, *European Symposium on Programming (ESOP'04)*, volume 2986, pages 33–48, 2004.

[GB06] Laurent Granvilliers and Frédéric Benhamou. Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156, 2006.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. pages 627–633, 06 2009.

[GH11] Diarmuid Grimes and Emmanuel Hebrard. Models and strategies for variants of the job shop scheduling problem. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 356–372. Springer-Verlag, 2011.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 18–34, 2006.

[GP08] Eric Goubault and Sylvie Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. 08 2008.

[GPfP06] Ian P. Gent, Karen E. Petrie, and Jean françois Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[Gra92] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1992.

[Gra97] Philippe Granger. Static analyses of congruence properties on rational numbers (extended abstract). In Pascal Van Hentenryck, editor, *Static Analysis*, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Han92] Eldon Hansen. *Global optimization using interval analysis*. Marcel Dekker, 1992.

[HE79] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial intelligence (IJCAI'79)*, pages 356–364. Morgan Kaufmann Publishers Inc., 1979.

[HK14] Kunihito Hoki and Tomoyuki Kaneko. Large-scale optimization for evaluation functions with minimax search. *J. Artif. Int. Res.*, 49(1):527–568, January 2014.

[JKDW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer London Ltd, August 2001. http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-1-4471-1067-5.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21th International Conference Computer Aided Verification (CAV 2009)*, 2009.

[JW93] Luc Jaulin and Eric Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. volume 1, pages 390–396, 01 2005.

[KDG04] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2004.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: Compcert and the C standard. In *ITP*, volume 8558 of *Lecture Notes in Computer Science*, pages 543–548. Springer, 2014.

[LCM18] Fanghui Liu, Waldemar Cruz, and Laurent Michel. A complete tolerant algebraic side-channel attack for aes with cp. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 259–275, Cham, 2018. Springer International Publishing.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[Liu98] Baoding Liu. Minimax chance constrained programming models for fuzzy decision systems. *Information Sciences*, 112(1):25 – 38, 1998.

[LR19] Christophe Lecoutre and Olivier Roussel. Proceedings of the 2018 XCSP3 competition. *CoRR*, abs/1901.01830, 2019.

[LV92] Hervé Le Verge. A Note on Chernikova's algorithm. Research Report RR-1662, INRIA, 1992.

[MBR16] A. Miné, J. Breck, and T. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In *Proc. of the 25rd European Symposium on Programming (ESOP'16)*, volume 9632 of *Lecture Notes in Computer Science (LNCS)*, pages 560–588. Springer, Apr. 2016. http://www-apr.lip6.fr/~mine/publi/article-mine-al-esop16.pdf.

[MFK+16] Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 166–184, 2016.

[MGM15] Morten Mossige, Arnaud Gotlieb, and Hein Meling. Testing robot controllers using constraint programming and continuous integration. *Information and Software Technology*, 57:169 – 185, 2015.

[Min01] Antoine Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 310–, Washington, DC, USA, 2001. IEEE Computer Society.

[Min02] Antoine Miné. A Few Graph-Based Relational Numerical Abstract Domains. LNCS 2477, pages 117–132. Springer, September 2002.

[Min04] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure, December 2004.

[Min06a] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[Min06b] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.

[Min07] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. *CoRR*, abs/cs/0703076, 2007.

[Min15] Antoine Miné. AstréeA: A Static Analyzer for Large Embedded Multi-Task Software. In *16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931 of *Lecture Notes in Computer Science*, page 3, Mumbai, India, January 2015. Springer.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[MOJ18] A. Miné, A. Ouadjaout, and M. Journault. Design of a Modular Platform for Static Analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), page 4, 28 Aug. 2018. http://www-apr.lip6.fr/~mine/publi/mine-al-tapas18.pdf.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[Mon16] David Monniaux. A Survey of Satisfiability Modulo Theory. In *Computer Algebra in Scientific Computing*, Bucharest, Romania, September 2016.

[Moo66] Ramon Edgar Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs N. J., 1966.

[MRL01] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *CP*, 2001.

[Nel80] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[NGVR12] Vincent Noel, Dima Grigoriev, Sergei Vakulenko, and Ovidiu Radulescu. Tropical geometries and dynamics of biochemical networks application to hybrid cell cycle models. *Electronic Notes in Theoretical Computer Science*, 284:75 – 91, 2012. Proceedings of the 2nd International Workshop on Static Analysis and Systems Biology (SASB 2011).

[Pel12] Marie Pelleau. *Domaines abstraits en programmation par contraintes*. PhD thesis, Université de Nantes, November 2012.

[Pel15] Marie Pelleau. 2 - abstract interpretation for the constraints. In Marie Pelleau, editor, *Abstract Domains in Constraint Programming*, pages 53 – 68. Elsevier, 2015.

[PFL14] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[PMTB13a] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 434–454, January 2013.

[PMTB13b] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, 2013.

[PRP16] Alexandre Papadopoulos, Pierre Roy, and François Pachet. Assisted lead sheet composition using flowcomposer. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 769–785, Cham, 2016. Springer International Publishing.

[PTB14] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. The octagon abstract domain for continuous constraints. *Constraints*, 19(3):309–337, 2014.

[Pug98] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National/10th Conference on Artificial Intelligence/Innovative applications of artificial intelligence (AAAI '98/IAAI '98)*, pages 359–366. American Association for Artificial Intelligence, 1998.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, pages 337–351, 1982.

[Ram97] Edgar A. Ramos. Construction of 1-d lower envelopes and applications. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pages 57–66, New York, NY, USA, 1997. ACM.

[Rat94] Dietmar Ratz. Box-splitting strategies for the interval Gauss-Seidel step in a global optimization method. *Computing*, 53:337–354, 1994.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. pages 557–571, 09 2004.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

[Sch02] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[SD07] Jean Souyris and David Delmas. Experimental assessment of astrée on safety-critical avionics software. In *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security*, pages 479–490, 2007.

[SS08] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.

[SSY15] Shweta Soner, Swapnil Soner, and Maya Yadav. A survey on software bug evaluation. *International Journal of Computer Applications*, 129:36–38, 11 2015.

[Tal18] Pierre Talbot. *Spacetime Programming: A Synchronous Language for Constraint Search*. PhD thesis, Sorbonne University, 2018.

[TC07] Gilles Trombettoni and Gilles Chabert. Constructive interval disjunction. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 635–650, 2007.

[vHMD97] Pascal van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: a Modeling Language for Global Optimization*. MIT Press, 1997.

[vHYGD08] Pascal van Hentenryck, Justin Yip, Carmen Gervet, and Grégoire Dooms. Bound consistency for binary length-lex set constraints. In *Proceedings of the 23rd National Conference on Artificial intelligence (AAAI'08)*, pages 375–380. AAAI Press, 2008.

[Vio07] Julien Vion. Hybridation de prouveurs CSP et apprentissage. In *Troisièmes Journées Francophones de Programmationpar Contraintes (JFPC07)*, JFPC07, INRIA, Domaine de Voluceau, Rocquencourt, Yvelines France, June 2007.

[Wal06] Toby Walsh. General symmetry breaking constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, CP'06, pages 650–664, Berlin, Heidelberg, 2006. Springer-Verlag.

[YBR$^+$19] Tomoya Yamaguchi, Martin Brain, Chirs Ryder, Yosikazu Imai, and Yoshiumi Kawamura. Application of abstract interpretation to the automotive electronic control system. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 425–445, Cham, 2019. Springer International Publishing.

[ZMRM17] Heytem Zitoun, Claude Michel, Michel Rueher, and Laurent Michel. Search strategies for floating point constraint systems. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 707–722, Cham, 2017. Springer International Publishing.

[ZP09] Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. *Constraints*, 14(3):392–413, Sep 2009.

[ZPTM18] Ghiles Ziat, Marie Pelleau, Charlotte Truchet, and Antoine Miné. Finding solutions by finding inconsistencies. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 420–435, Cham, 2018. Springer International Publishing.

Appendix A

Improvement to the AbSolute Solver

Abstract

AbSolute is a constraint solver based on abstract domains from the theory of abstract interpretation and has served as a practical basis for experiments in this thesis. It features a modular and functional OCaml implementation and is available through the opam package manager. Its tunable solving method in terms of precision, heuristics, propagation loop, output formatting and abstract domain makes it able to solve a large variety of constraint satisfaction problems. During this thesis, we have developed some techniques to improve the performances of the solver. Also, we have augmented its capabilities with a lot a features, some of which we did not have the opportunity to introduce in the previous chapters. This annex is meant to briefly address these points and discuss the implementation aspects of this work.

A.1 Problem description language

To describe constraint satisfaction problems, AbSolute features a description language which we present in this section. It allows the user to declare variables, and constraints, and annotate the problem with comments easily in text file, as illustrated in example A.1.

```
/* A polynomial constraint over two variables  */
init{
  real x = [-10;10];
  real y = [-5;5];
}
constraints{
  (x-2)^2 + (y-2)^2 <= 4;
}
```

Listing A.1: a simple example of CSP using AbSolute's description language

Problems are divided in two parts: the variables declaration in the **init** tag and the constraints declaration in the **constraint** tag. Variable are described according to their type (integer or real variables) and their range. Constraints are enumerated in sequence and interpreted conjunctively A third optional tag can be added to describe the results of the problem. This description will be used to verify the validity of the implementation as we will see in the section A.4.

A.1.1 The grammar of the language

We present now the grammar of the language in its *BNF* form. Constraints are Boolean expression over the variables of the problem. They manipulates arithmetical expressions as explicated in figure A.1.

<bool-expr>	::=	<arith-expr> $\square$ <arith-expr>	$\square \in \{>, \geq, <, \leq, =, \neq\}$
	$\mid$	$\neg$ <bool-expr>	negation
	$\mid$	<bool-expr> $\vee$ <bool-expr>	disjunction
	$\mid$	<bool-expr> $\wedge$ <bool-expr>	conjunction

Figure A.1: Boolean expression syntax

Arithmetic expressions, whose grammar is given in table A.2 include constants, variables, usual operators and function calls among a list of predefined functions listed in table A.1.

<arith-expr>	::=	c	$c \in \mathbb{R}$
	$\mid$	$\mathcal{V}$	variables
	$\mid$	<arith-expr> $\diamond$ <arith-expr>	$\diamond \in \{+, -, *, /, \%\}$
	$\mid$	- <arith-expr>	opposite
	$\mid$	*ident* '('<arith-expr>(',' <arith-expr>)*')'	function calls

Figure A.2: Arithmetical expression syntax

Available functions

We have extended the standard arithmetic with several functions that make the constraint language more practical and allow us to tackle more problem. Those are:

Name	Description
max	maximum value of two numbers
min	minimum value of two numbers
sqrt	Square root of a number
ln	Logarithm of a number
exp	Image of a number with respect to the exponentiation
cos	Cosine of a number
sin	Sinus of a number
acos	Arc cosine of a number
asin	Arc sinus of a number
tan	Tangent of a number
atan	Arc tangent of a number

Table A.1: Available functions in the constraint language of AbSolute

A.2 Global architecture of the solver

The AbSolute solver is implement in a generic way at several level. The main solving loop, based on propagation and exploration is completely independent from the abstract domains, these one being chosen dynamically, at solving time. In the same fashion, the abstract domains are built as separate units defining a representation and providing the necessary operations over this representation for the resolution : split, size, propagation etc. By the end of the solving process, the solver returns a collection of abstract element, tagged either as sure elements, *i.e.* satisfying the constraints, or as unsure, *i.e.* not necessarily satisfying the constraint. Also, the implementation of different abstract domains maintains this modularity and factorizes the common treatment of the different domains. For example, the propagation of Boolean expressions is made transparently, without having to know the numerical underlying abstract domain. From there, the different abstract domains are separated into two categories: the relational and the non-relational abstract domains. The relational abstract, *i.e.* the Octagon and Polyhedra reuses the features of Apron to define abstraction for numerical expressions (no Boolean expression are allowed). We define the split, size and propagators in a generic way using a decomposition of the abstract value into a set of linear constraints or a set of generators. The non-relational abstract domains, *i.e.* intervals and congruences are handled within a Cartesian representation of the search space, where for each variable is associated a representation. Here again, the variable selection, filtering and evaluation do not depend on the representation used for the variables. This allows us to have easily several kinds of non-relational abstract domains as soon as we are able to define the required operations for the used representation. Lastly, as we use several kind of intervals (real and discrete), with different types of bounds, (integer, floats and rationals), we have defined an interval arithmetic parametrized by the type of bound being used and once again, our interval do not depend on the representation used for the bounds. Those only have to fulfill some requirement, such as providing arithmetic operations with a sound handling of rounding errors. Finally, all of these abstract domains can be combined into domain products, and be used transparently, as we provide for the different possible combinations the associated reduction operation that ensure the communication between the two components. Figure A.3 summarizes the hierarchy of the different units mentioned above.

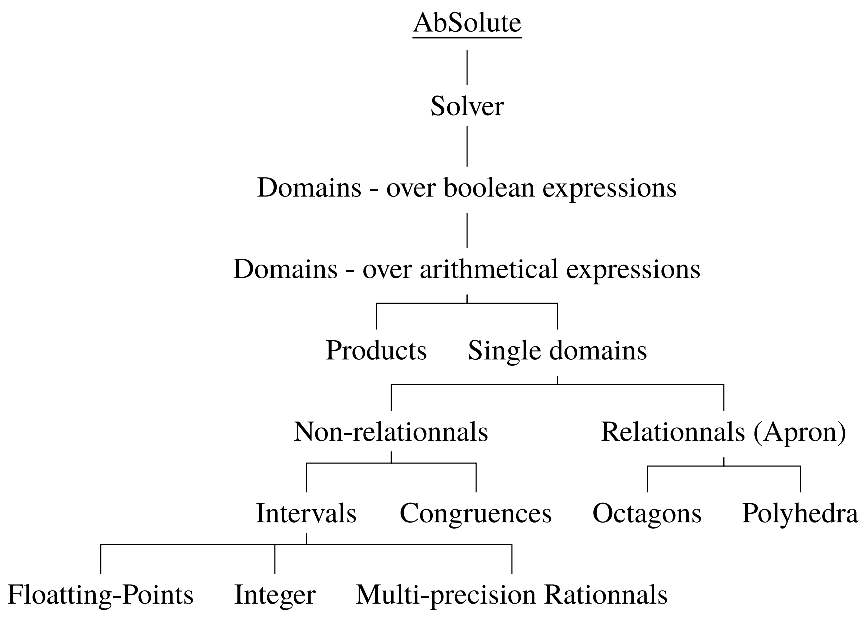


Figure A.3: Hierarchy of the different abstract domains of AbSolute

A.3 Visualization

We have implemented several visualization means inside AbSolute to be able to have a graphical representation of the solution space. This tools were a key feature to perform a quick, even though not very rigorous, testing of the solver soundness. It also gives a more comprehensive output than a textual output.

A.3.1 2-dimensions visualization

The implementation of a visualization interface of abstract elements follows the same principle of modularity than the rest of the solver. The visualization tool does not need to know the representation of the abstract domain being used. It only requires a set of operation to be available in the abstract such as a conversion of element into a list of shapes to draw. In the case where the problem contains more than two variables, it also needs a projection function for our abstract elements: the solution space is projected on the first two variables of the problem in the order of their declaration in the problem. This process is shared between all of our different visualization back-ends.

A.3.2 OCaml Graphics

The graphical interface provided with AbSolute is launched automatically at the end of the resolution when the *-v* is specified option on the command line. It is based on the Graphics module of OCaml and displays all the abstract elements returned by the solver. In the case where the problem contains more than two variables, the solution space is projected on the first two variables of the problem in the order of their declaration. Figure A.4, illustrates this interface.

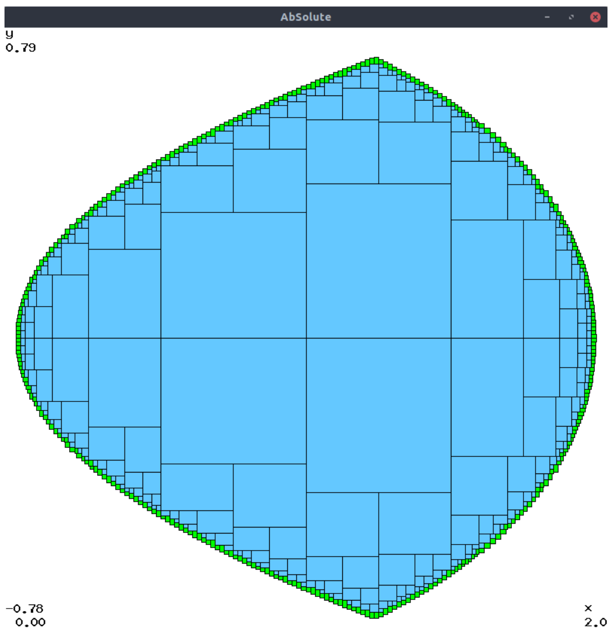


Figure A.4: AbSolute's 2D graphical output of the solutions of the problem 22

A.3.3 Latex

AbSolute also features another visualization back-end which is the **PGF/TikZ**[1]. This is a pair of languages for producing vector graphics from a geometric/algebraic description. It allows the solver to generate a **latex** code corresponding to the drawing of the solution space, as illustrated in Figure 23.

```
\documentclass{standalone}

\usepackage{xcolor}
\usepackage{pgf, tikz}

\definecolor{sure}{rgb}{0.4, 0.8, 1}
\definecolor{unsure}{rgb}{0, 0.9, 0}

\begin{document}
  \begin{tikzpicture}
    \draw [fill=sure] (0.156, −0.265) rectangle (0.250, −0.176);
    \draw [fill=sure] (0.070, −0.176) rectangle (0.125, −0.088);
    \draw [fill=sure] (0.062, −0.088) rectangle (0.125, 0.000);
    \draw [fill=sure] (0.125, −0.176) rectangle (0.250, 0.000);
    ...
    \draw [fill=unsure] (0.104, −0.279) rectangle (0.156, −0.228);
    \draw [fill=unsure] (0.062, −0.228) rectangle (0.156, −0.176);
    \draw [fill=unsure] (0.156, −0.353) rectangle (0.250, −0.265);
    \draw [fill=unsure] (0.015, −0.176) rectangle (0.070, −0.088);
    ...
  \end{tikzpicture}
\end{document}
```

Listing A.2: **TikZ** code generation of the problem 22

A.3.4 3D

To be able to have a more meaningful visualization for problems with a lot of variables, we have implemented in AbSolute an **OBJ** file output. **OBJ** is a file format containing the description of a 3D geometry. Geometric shapes can be defined by polygons or smooth surfaces such as rational and non-rational surfaces. Each surface is described by a set of vertices (accompanied by texture coordinates and normals at each vertex) and a set of faces. To be able to visualize our solutions sets in this format, we have to convert abstract element into vertices and build the faces between those vertices. Figures A.5 and A.6 show screen-shots of this visualization using a viewer of **OBJ** files.

Figure A.7 summarizes the hierarchy of the different tools of visualization of AbSolute.

[1]The reader of this manuscript having a minimal knowledge of TikZ has probably already understood this from the figure5.1

Figure A.5: Different views of the 3D OBJ output of the solutions of the problem 23

Figure A.6: Different views of the 3D OBJ output of the solutions of the problem 24

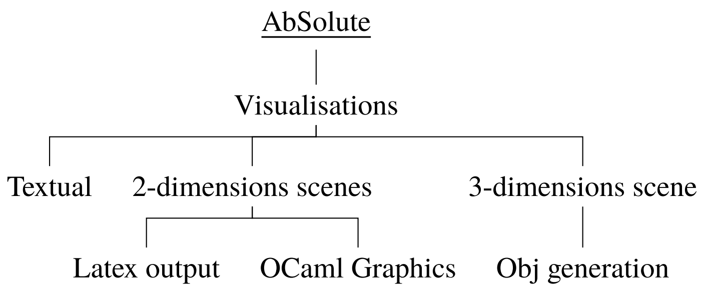


Figure A.7: Hierarchy of the different visualization tools of AbSolute

A.4 Testing framework

We have incorporated into the solver a testing framework to improve the robustness of the implementation. This testing framework uses two different methods to check the validity of the implementation: First, using extra-information of a problem, if present, it verifies some properties about the results. And second, it generates randomly instance from the results of the solver and then verifies if these are indeed solutions of the constraint problem.

A.4.1 From the annotation of a problem:

The extra-information of a problem come in the form of an additional *solutions* tag, in which we can specify four possible annotations:

- Instances that are solutions. We make sure that these belong to an abstract element of the partition returned by the solver. This is a completeness test.

- Instances that do not satisfy at least one constraint. We make sure that these do not belong to any *sure* element of the partition. This is a soundness test.
- The user can also add the *unfeasible* annotation to specify that the given problem admits no solution.
- The *tautology* annotation can be used to specify that the problem admits only solution within the declared domains.

Also, the testing framework ensures that all of the instances specified as solutions in the annotations belong to one and only one abstract element. This is an irredundancy test. Figure A.8 illustrates the syntax for the annotations.

```
/* exponential test */

init {
  real x = [-100;100];
  real y = [-100;100];
}

constraints {
  exp(x) = y;
}

solutions {
  {y=1;x=0};
  !{y=1;x=0.001};
  !{y=0.99;x=0}
}
```

Listing A.3: a simple example of CSP using AbSolute's description language

Figure A.8: Example of annotation used by the testing framework of AbSolute.

A.4.2 From the solver's output

The second method used by the solver to test the validity of the result is based on a random generation of instances within the abstract elements of the cover of the solution space. For every element tagged as *sure i.e.* satisfying the constraints, we generate randomly a certain number of instances and verify that they are indeed solutions of the problem. This requires our abstract elements to define a random generation procedure, that we detail now.

Random generators

For boxes. As boxes are Cartesian product of interval, simply generate a for each variable a uniformly randomly chosen value from the interval. This generation is lifted easily to the Cartesian representation.

For polyhedra The random generation of a point within a bounded polyhedra, in a uniform way, is not as easy. Of course, we can use a rejection sampling by generating point with the minimum enclosing box of a polyhedron and reject them if the obtained instance does not belong to the polyhedron. Repeating this step until a point inside the polyhedron is found is correct, when this algorithm terminates. The problem is that the complexity of this procedure is proportional to the ratio $\frac{volume_{box}}{volume_{poly}}$, and in particular, in presence of equalities in the polyhedron, this ratio is infinite and this algorithm do not terminate in the general case. To address this problem, we have made the choice of fixing a variable to one value of its range after fixed number of tries. We repeat this process with the polyhedron where this variable has been removed. This is giving-up on uniformity to enforce termination.

APPENDIX B

CONSTRAINT SATISFACTION PROBLEM EXAMPLES

This section presents some of the problems used to illustrate the work we have done in this thesis. Problems are presented according to their name, number of variables and their type (discrete or continuous), and number of constraints. Extra information about the resolution are then given.

Example 20. *An example with 2 variables $(x, y) \in \mathbb{R}^2$ subject to 1 constraint.*

- $x \in [1; 50]$
- $y \in [-1.5; 1]$

Constrained with:
$\cos(ln(x)) > y$

Abstract domain used	*Box*
Inner solutions	*10193*
Inner volume	*37.29*
Outer solutions	*8192*
Outer volume	*0.36*
Inner ratio	*99.03*
Solving time	*0.13s*
Precision	*0.01*

Example 21. *A simple example with 2 variables $(x, y) \in \mathbb{R}^2$ subject to 2 trigonometrical constraints.*

- $x \in [-10; 10]$
- $y \in [-4; 4]$

Constrained with:

- $y < \sin(x) + 1$
- $y > \cos(x) - 1$

Abstract domain used	*Box*
Inner solutions	*6832*
Inner volume	*40.82*
Outer solutions	*6528*
Outer volume	*0.37*
Inner ratio	*99.09*
Solving time	*0.13s*
Precision	*0.01*

Example 22. *Mickey problem from the Coconut benchmark: 2 variables* $(x, y) \in \mathbb{R}^2$ *subject to 2 constraints.*

- $x \in [-3; 3]$
- $y \in [-3; 3]$

Constrained with:

- $((2. * (y^2.)) - x) <= 0.$
- $(((x^2.) + (4. * (y^2.))) - 4.) <= 0.$

Abstract domain used	*Box*
Inner solutions	*356*
Inner volume	*2.082*
Outer solutions	*358*
Outer volume	*0.06*
Inner ratio	*96.84*
Solving time	*0.019s*
Precision	*0.02*

Example 23. *Spoon problem: 3 variables* $(x, y, z) \in \mathbb{R}^3$ *subject to 1 constraint.*

- $x \in [-3; 3]$
- $y \in [-3; 3]$
- $z \in [-3; 3]$

Constrained with:
$((3 * x^2 + (y - 1.9)^2 + (2 * z)^2 - 1)^2 + (0.2 * z)) * (((((0.8 * z + 1.2)^3 + (5 * y - 6))^2 + (4 * x)^2 - 0.5) * ((x)^2 + (y + 6)^2 + (z - 2.8)^2 - 0.3) * (x^2 + (y - 1)^2 + (z + 3.3)^2 - 0.03) + 290) * (9 * x^2 + (y - 0.1 * z + 2.5)^2 + (4 * z - 5 + y)^2 - 1) - 400) - 99 = 0$

Abstract domain used	*Box*
Inner solutions	*0*
Inner volume	*0*
Outer solutions	*356098*
Outer volume	*0.09*
Inner ratio	*0*
Solving time	*72.62s*
Precision	*0.01*

Example 24. *Diabolo problem: 3 variables* $(x, y, z) \in \mathbb{R}^3$ *subject to 1 constraint.*

- $x \in [-5; 5]$
- $y \in [-5; 5]$
- $z \in [-5; 5]$

Constrained with:
$x^2 = (y^2 + z^2)^2$

Abstract domain used	*Box*
Inner solutions	*0*
Inner volume	*0*
Outer solutions	*470720*
Outer volume	*1.42*
Inner ratio	*0*
Solving time	*11.37s*
Precision	*0.02*

Abstract

We investigate in this thesis a tight collaboration between techniques of Abstract Interpretation and Constraint Programming within a unified method of resolution of constraint satisfaction problems. This work addresses the problem of the design in a generic and modular way of a constraint solver based on abstract domains, which capture specific properties of program or constrained variables. We exploit the assets of both fields to bypass the restriction of standard constraint solvers, such as the dedication to a certain type of constraints or variables. Our effort also consists in the design of a robust method, providing soundness properties even in the context of floating-point errors. Moreover, we are interested in different techniques allowing the construction of a partition of a solution space that can be easily reused, both from a quantitative and a qualitative point of view. Our work has been concretized in the form of an implementation within the AbSolute constraint solver and applied on several examples. Our experiments show that our methods improve the solver's efficiency or the quality of the results according to the metrics we have defined.

Résumé

Nous étudions dans cette thèse une collaboration étroite entre les techniques de l'Interpretation Abstraite et de la Programmation Par Contraintes au sein d'une méthode unifiée de résolution de problèmes de satisfaction de contrainte. Ce travail aborde le problème de la conception de manière générique et modulaire d'un solveur de contraintes basé sur des domaines abstraits qui permettent l'inférence de propriétés spécifiques d'un programme ou d'un système de contraintes. Nous exploitons les atouts des deux domaines pour contourner les restrictions des solveurs de contraintes standards, telles que la spécialisation à un certain type de contraintes ou de variables. Notre travail consiste aussi à concevoir une méthode robuste tenant compte des problématiques liées à l'utilisation de calculs en précision flottante avec une gestion correcte des erreurs d'arrondi. De plus, nous nous intéressons à différentes techniques permettant la construction d'une partition d'un espace de solution qui peut être facilement réutilisée, tant d'un point de vue quantitatif que qualitatif. Notre travail a été concrétisé sous la forme d'une implémentation dans le solveur de contraintes AbSolute et appliqué sur plusieurs exemples. Les expériences que nous avons menées au sein de ce solveur montrent que nos méthodes améliorent l'efficacité du solveur et la qualité de ses résultats par rapport aux métriques que nous avons defini.