A Constraint Solver based on Abstract Domains

Marie Pelleau†, Antoine Miné‡, Charlotte Truchet† and Frédéric Benhamou†

†Université de Nantes
2 rue de la Houssinière, Nantes
{firstName.lastName}@univ-nantes.fr

‡École normale supérieure
45, rue d'Ulm, Paris
Antoine.Mine@ens.fr

Abstract. In this article, we apply techniques from Abstract Interpretation (a general theory of semantic abstractions) to Constraint Programming (which aims at solving hard combinatorial problems with a generic framework based on first-order logics). We highlight some links and differences between these fields: both compute fixpoints by iteration but employ different extrapolation and refinement strategies; moreover, consistencies in Constraint Programming can be mapped to non-relational abstract domains. We then use these correspondences to build an abstract constraint solver that leverages abstract interpretation techniques (such as relational domains) to go beyond classic solvers. We present encouraging experimental results obtained with our prototype implementation.

1 Introduction

Abstract Interpretation is a method to design approximate semantics of programs and provide sound answers to questions about their run-time behaviors [8,7]. Constraint Programming aims at solving, with reusable techniques, hard combinatorial problems expressed declaratively. This article studies the application of Abstract Interpretation techniques to Constraint Programming.

State of the art. First introduced by Montanari [19], Constraint Programming (CP) relies on the idea that many problems can be expressed as conjunctions of first-order logic formulas, called constraints, each one representing a specific combinatorial feature of the problem [21]. Each constraint comes with *ad hoc* operators exploiting its internal structure to reduce the combinatorics. The constraints are then combined into generic solving algorithms. Much of the research effort in CP is focused on defining and improving constraints[1] and fine-tuning solving algorithms. CP now offers powerful techniques for combinatorial optimization, with many practical applications to scheduling, packing, layout design, frequency allocation, etc. Yet, solvers suffer from limitations. They are limited to non-relational domains, such as boxes or Cartesian products of integer sets. Moreover, two clearly separate family of solving algorithms exist: one handles

[1] See `http://www.emn.fr/z-info/sdemasse/gccat` for a catalog of existing global constraints.

discrete variables and the other continuous variables. Several methods have been proposed to handle mixed problems, such as discretizing continuous variables and handling them in a discrete solver (as in Choco [23]). Unfortunately, the solver remains a purely discrete one and does not benefit from heuristics developed for continuous ones. Alternatively, one can add specific mixed constraints [6] or generic integrity constraints [4] to a continuous solver, with similar drawbacks.

In another research area, Abstract Interpretation (AI) is used to design static program analyzers that are sound and always terminate (such as Astrée [5]) by developing computable approximations of essentially undecidable problems. The (uncomputable) concrete collecting semantics expresses in fixpoint form the set of observable behaviors of the program. It is approximated in an abstract domain that restricts the expressiveness to a set of properties of interest, provides data-structure representations, efficient algorithms to compute abstract versions of concrete operators, and acceleration operators to approximate fixpoints in finite time. Soundness guarantees that the analyzer observes a super-set of the program behaviors. Numeric domains, focusing on numeric variables and properties, are particularly well developed; major ones include intervals [7] and polyhedra [9], and recent years have seen the development of new domains, such as octagons [18], and libraries, such as Apron [15]. They can handle all kinds of numeric variables, including mathematical integers, rationals, and reals, machine integers and floating-point numbers, and even express relationships between variables of different types [17,5]. Each domain corresponds to some trade-off between cost and precision. Finally, domains can be modified and combined by generic operators, such as disjunctive completions and reduced products.

Contribution. In this paper we seek to use AI techniques to build an abstract, generic CP solver. Our contributions are as follows: we show the links between AI and CP and recast the later as a fixpoint computation similar to local iterations in a disjunctive completion of non-relational domains; we design a generic abstract solver parametrized by abstract domains and prove its termination; we show that, by using relational and mixed integer-real abstract domains, we can go beyond some limitations of existing solvers. We do not study in this paper the dual problem, *i.e.,* exploiting CP techniques in AI; it is one of the perspectives of this work.

The paper is organized as follows. Section 2 provides background information on AI and CP, and some elements of comparison. Section 3 recasts CP as AI and presents our abstract solver. Our prototype implementation and preliminary experimental results are presented in Sec. 4. Section 5 concludes.

Related works. Some interactions between CP and verification techniques have been explored in previous works. For instance, CP has been used to automatically generate test configurations [13], or to verify CP models [16]. In another direction, several recent works, such as [10,24], establish connections between AI and SAT solving algorithms, holding promise for cross-pollination between these fields. Our aim is similar, but linking AI to CP. While related, CP and SAT differ significantly enough in the chosen models (numeric versus boolean) and solving algorithms that previous results do not apply. Our work is in the continuity of [25] that extends CP solving methods to use richer domain representations, such as octagons. However, embedding a new domain required *ad hoc* techniques to express its operations in the native language of the solver: boxes. In this paper, we reverse that process: we redesign from the ground up the solver in an abstract way so that it is not tied to boxes but can reuse as-is existing abstract operators and domains from AI.

2 Preliminaries

In this section we present some notions of Abstract Interpretation and Constraint Programming that will be needed later.

2.1 Bases of Abstract Interpretation

We first present some elements of Abstract Interpretation that will prove useful in the design of our solver (see [8,7] for a more detailed presentation).

Fix-point abstractions. The concrete semantics of a program is given as the least fixpoint $\mathrm{lfp}_\perp F$ of an operator $F : \mathcal{D} \to \mathcal{D}$ in some partially ordered structure $(\mathcal{D}, \sqsubseteq, \perp, \sqcup)$, such as a complete partial order or a lattice. With suitable hypotheses [8] on F and $\mathcal{D}$, the fixpoint can be expressed as the limit of a (possibly transfinite) increasing iteration $\mathrm{lfp}_\perp F = \bigsqcup_{i \in \mathrm{Ord}} F^i(\perp)$ on ordinals.

Similarly, we denote by $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \perp^\sharp, \sqcup^\sharp)$ the abstract domain. A monotonic concretization $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ associates a concrete meaning to each abstract element. An abstract operator $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a sound abstraction of F if $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$. Sometimes, but not always, there exists an abstraction function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ such that (α, γ) forms a Galois connection, which ensures that each concrete element X has a best abstraction $\alpha(X)$, and the optimal abstract operator $F^\sharp$ can be uniquely defined as $F^\sharp = \alpha \circ F \circ \gamma$. In all cases, $\mathrm{lfp}_\perp F$ can be approximated as $\bigsqcup^\sharp_{i \in \mathrm{Ord}} F^{\sharp i}(\perp^\sharp)$. This limit may not be computable, even if $F^\sharp$ is, or may require many iterations. It is thus often replaced with the limit of an increasing sequence: $X_0^\sharp = \perp^\sharp$, $X_{i+1}^\sharp = X_i^\sharp \triangledown F^\sharp(X_i^\sharp)$ using a widening operator $\triangledown$ to accelerate convergence. The widening is designed to over-approximate $\sqcup$ and converge in finite time δ to a post-fixpoint $X_\delta^\sharp$ of $F^\sharp$. Then, $\gamma(X_\delta^\sharp) \sqsupseteq \mathrm{lfp}_\perp F$. The limit is often refined by a decreasing iteration: $Y_0^\sharp = X_\delta^\sharp$, $Y_{i+1}^\sharp = Y_i^\sharp \vartriangle F^\sharp(Y_i^\sharp)$, using a narrowing operator $\vartriangle$ designed to stay above any fixpoint of F and converge in finite time. As all the $Y_i^\sharp$ are abstractions of $\mathrm{lfp}_\perp F$, we can stop the iteration at any time.

Local iterations. In addition to refining the results of least fixpoint computations, decreasing iterations have been used by Granger [11] locally, *i.e.,* within the computation of $F^\sharp$. Granger observes that the concrete operator F often involves lower closure operators, *i.e.,* operators ρ that are monotonic, idempotent ($\rho \circ \rho = \rho$) and reductive ($\rho(X) \sqsubseteq X$). Given any sound abstraction $\rho^\sharp$ of ρ, the limit $Y^\sharp_\delta$ of the sequence $Y^\sharp_0 = X^\sharp$, $Y^\sharp_{i+1} = Y^\sharp_i \vartriangle \rho^\sharp(Y^\sharp_i)$ is an abstraction of $\rho(\gamma(X^\sharp))$. Whenever $\rho^\sharp$ is not an optimal abstraction of ρ, $Y^\sharp_\delta$ may be significantly more precise than $\rho^\sharp(X^\sharp)$. A relevant application is the analysis of complex test conjunctions $C_1 \wedge \cdots \wedge C_p$ where each atomic test C_i is modeled in the abstract as $\rho^\sharp_i$. Generally, $\rho^\sharp = \rho^\sharp_1 \circ \cdots \circ \rho^\sharp_p$ is not optimal, even when each $\rho^\sharp_i$ is. A complementary application is the analysis of a single test C_i using a sequence of relaxed, non-optimal test abstractions. For instance, non-linear expression parts may be replaced with intervals computed based on variable bounds [17]. As applying the relaxed test refines these bounds, the relaxation is not idempotent and benefits from local iterations. The link between local iterations and least fixpoint refinements lies in the observation that $\rho(X)$ computes a trivial fixpoint: the greatest fixpoint of ρ smaller than X: $\mathrm{gfp}_X\, \rho$. In both cases, a decreasing iteration starts from an abstraction of a fixpoint ($\mathrm{lfp}_\perp\, F$ in one case, $\mathrm{gfp}_X\, \rho$ in the other) and computes a smaller abstraction of that fixpoint.

On narrowings. While a lot of work has been devoted to designing smart widenings, narrowings have gathered far less attention. Some major domains, such as polyhedra [9], do not feature any. This may be explained by three facts: firstly, narrowings (unlike widenings) are not necessary to achieve soundness; secondly, performing a bounded number of decreasing iterations without narrowing is sometimes sufficient to recover enough precision after widening [5]; thirdly, when this simple technique is not sufficient, narrowings do not actually help further in practice and solutions beyond decreasing iterations must be considered [12]. In the following, we argue that Constraint Programming can be seen as a form of decreasing iteration, but uses different techniques that are, in some respects, more advanced than the corresponding ones used in Abstract Interpretation.

2.2 Constraint Programming

We now present the basic definitions of Constraint Programming (see [21] for a more detailed presentation). In this section, we employ CP terminology, and take special care to point out terms with a different meaning in AI and CP.

Problems are modeled in a specific format, called Constraint Satisfaction Problem (CSP), and defined as follows:

Definition 1 (Constraint Satisfaction Problem). *A CSP is defined by a set of variables* $(v_1, \ldots, v_n)$ *taking their value in domains* $(\hat{D}_1, \ldots, \hat{D}_n)$ *and a set of constraints* $(C_1, \ldots, C_p)$ *that are relations on the variables.*

A *domain* D_i in CP denotes the set of possible values for a variable v_i and $D = D_1 \times \cdots \times D_n$ is called the *search space.* As the search space evolves during the solving process, we distinguish the initial search space of the CSP and note it $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ as in Def. 1. Problems may be discrete ($\hat{D} \subseteq \mathbb{Z}^n$) or continuous ($\hat{D} \subseteq \mathbb{R}^n$). Domains are, however, always bounded.

Given a constraint C on variables $v_1, \ldots, v_n$ in domains $D_1, \ldots, D_n$, and given values $x_i \in D_i$, we denote by $C(x_1, \ldots, x_n)$ the fact that the constraint is satified when each variable v_i takes the value x_i. The set of solutions is $S = \{(s_1, \ldots, s_n) \in \hat{D} \,|\, \forall i \in [\![1, p]\!],\, C_i(s_1, \ldots, s_n)\}$, with p the number of constraints and where $[\![a, b]\!] = \{x \in \mathbb{Z} \,|\, a \leq x \leq b\}$ denotes the interval of integers between a and b.

For discrete problems, two domain representations are traditionally used: subsets and intervals.

Definition 2 (Integer Cartesian Product). *Let $v_1, \ldots, v_n$ be variables over finite discrete domains $\hat{D}_1, \ldots, \hat{D}_n$. We call integer Cartesian product any Cartesian product of integer sets in $\hat{D}$. Integer Cartesian products form a finite lattice:*

$$\mathcal{S}^\sharp = \{\, \prod_i X_i \,|\, \forall i,\, X_i \subseteq \hat{D}_i \,\}$$

Definition 3 (Integer Box). *Let $v_1, \ldots, v_n$ be variables over finite discrete domains $\hat{D}_1, \ldots, \hat{D}_n$. We call integer box a Cartesian product of integer intervals in $\hat{D}$. Integer boxes form a finite lattice:*

$$\mathcal{I}^\sharp = \{\, \prod_i [\![a_i, b_i]\!] \,|\, \forall i,\, [\![a_i, b_i]\!] \subseteq \hat{D}_i,\, a_i \leq b_i \,\} \cup \{\emptyset\}$$

For continuous problems, domains are represented as intervals with floating-point bounds. Let $\mathbb{F}$ be the set of floating-point machine numbers. Given $a, b \in \mathbb{F}$, we note $[a, b] = \{x \in \mathbb{R} \,|\, a \leq x \leq b\}$ the interval of reals bounded by a and b, and $\mathbb{I} = \{[a, b] \,|\, a, b \in \mathbb{F}\}$ the set of such intervals.

Definition 4 (Box). *Let $v_1, \ldots, v_n$ be variables over bounded continuous domains $\hat{D}_1, \ldots, \hat{D}_n \in \mathbb{I}$. A box is a Cartesian product of intervals in $\hat{D}$. Boxes form a finite lattice:*

$$\mathcal{B}^\sharp = \{\prod_i I_i \,|\, \forall i,\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i\} \cup \{\emptyset\}$$

Solving a CSP means computing exactly or approximating its solution set S.

Definition 5 (Approximation). *A complete (resp. sound) approximation of the solution S is a collection $\mathcal{A}$ of domain sequences such that $\forall (D_1, \ldots, D_n) \in \mathcal{A}$, $\forall i$, $D_i \subseteq \hat{D}_i$ and $S \subseteq \bigcup_{(D_1, \ldots, D_n) \in \mathcal{A}} D_1 \times \cdots \times D_n$ (resp. $\bigcup_{(D_1, \ldots, D_n) \in \mathcal{A}} D_1 \times \cdots \times D_n \subseteq S$).*

Soundness guarantees that we find only solutions, while completeness guarantees that no solution is lost. On discrete domains, constraint solvers are expected to be sound and complete, *i.e.,* compute the exact set of solutions. This is generally impossible on continuous domains, and we usually withdraw either soundness (most of the time) or completeness. Note that the terms *sound* and *complete* have opposing definitions in AI and CP so, to avoid confusion, we will use the term *over-approximations* (resp. *under-approximations*) to denote CP-complete AI-sound (resp. CP-sound AI-complete) approximations.

In this article, we consider solving methods that over-approximate the solutions of continuous problems and compute the exact solutions of discrete ones. These methods alternate two steps: propagation and search.

Propagation. The goal of a propagation algorithm is to use the constraints to reduce the domains. Intuitively, we remove inconsistent values from domains, *i.e.,* values that cannot appear in any solution. Several definitions of consistency have been proposed in the literature. We present the most common ones.

Definition 6 (Generalized Arc-Consistency). *Given variables $v_1, \dots, v_n$ over finite discrete domains $D_1, \dots, D_n$, $D_i \subseteq \hat{D}_i$, the domains are said generalized arc-consistent (GAC) for a constraint C iff $\forall i \in [\![1, n]\!], \forall x_i \in D_i, \forall j \neq i, \exists x_j \in D_j$ such that $C(x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n)$ holds.*

Definition 7 (Bound-Consistency). *Given variables $v_1, \dots, v_n$ over finite discrete domains $D_1, \dots, D_n$, $D_i \subseteq \hat{D}_i$, the domains are said bound-consistent (BC) for a constraint C iff $\forall i \in [\![1, n]\!]$, D_i is an integer interval $[\![a_i, b_i]\!]$, and the condition of Def. 6 holds for $x_i = a_i$ and $x_i = b_i$ (but not necessarily other values of x_i in $[\![a_i, b_i]\!]$).*

Definition 8 (Hull-Consistency). *Given variables $v_1, \dots, v_n$ over continuous interval domains $D_1, \dots, D_n \in \mathbb{I}$, $D_i \subseteq \hat{D}_i$, the domains are said Hull-consistent for a constraint C iff $D_1 \times \cdots \times D_n$ is the smallest floating-point box containing all the solutions for C in $D_1 \times \cdots \times D_n$.*

Each constraint kind and consistency comes with an algorithm, called *propagator*, that tries to achieve consistency. When considering several constraints, a *propagation loop* iterates the constraint propagators until a fixpoint is reached. As shown in [2], the order of propagator applications does not matter, as the set of domains lives in a finite lattice ($\mathcal{B}^\sharp$, $\mathcal{I}^\sharp$ or $\mathcal{S}^\sharp$) and the consistent fixpoint is its unique least element. When consistency is too costly to achieve, the propagators and propagator loops settle instead for an over-approximation (*e.g.,* removing only some inconsistent values). In addition to providing a tighter search space, the propagation is sometimes able to discover that it contains no solution at all, or that all its points are solutions.

Search. Generally, propagation alone cannot compute the exact solution (in the discrete case) or a precise enough over-approximation (in the continuous case).

```
list of boxes sols ← ∅                                    /*stores the solutions*/
queue of boxes toExplore ← ∅                          /*stores the boxes to explore*/
push $\hat{D}$ in toExplore                   /*initialization with CSP search space*/

while toExplore ≠ ∅ do
    $b$ ← pop(toExplore)
    $b$ ← Hull-consistency($b$)
    if $b \neq \emptyset$ then
        if $b$ contains only solutions or $b$ is small enough then
            sols ← sols ∪ $b$
        else
            split $b$ into $b_1$ and $b_2$ by cutting in half along the largest box dimension
            push $b_1$ and $b_2$ in toExplore
```

Fig. 1. A classic continuous solver.

Thus, in a second step, a *search engine* is employed to try various assumptions on variable values. In the discrete case, a chosen variable is instantiated to each value in its domain. In the continuous case, its domain is split into two smaller subdomains. The solving algorithm continues by selecting a new search space and applying a propagation step (as it may no longer be consistent), and possibly making further choices. This interleaving of propagations and choices terminates when the search space can be proved to contain no solution, only solutions or, in the continuous case, when its size is below a user-specified threshold. In the discrete case, at worst, all the variables are instantiated. After exploring a branch, in case of failure or if all the solutions should be computed, the algorithm returns to a choice point (instantiation or split) by *backtracking* and tries another assumption.

We illustrate the search algorithm by an example solver in Fig. 1 corresponding to a continuous solver based on Hull-Consistency (Def. 8) computing an over-approximation of all the solutions. As explained above, a discrete solver would differ significantly. Existing solutions to embed discrete variables in continuous solvers consist in adding constraints expressing integerness and their propagators [6,4], while keeping a search engine based on continuous domains.

2.3 Comparing Abstract Interpretation and Constraint Programming

We now present informally some connections between Abstract Interpretation and Constraint Programming. The next section will make these connections formal by expression CP in the AI framework.

Both techniques are grounded in the theory of fixpoints in lattices. They pursue similar goals and means: computing or over-approximating solutions to complex equations by manipulating abstracted views of potential solution sets, such as boxes (called domains in CP, and abstract domain elements in AI). Their goals, however, do not coincide. Solvers aim at completeness and thus always allow refinement up to an arbitrary precision. On the contrary, the precision of abstract interpreters is fixed by their choice of abstract domains; they can seldom represent arbitrary precise over-approximations. AI embraces incompleteness. The choice of abstract domains sets the cost and precision of an interpreter, while the choice of domains sets the cost of a solver to reach a given precision.

Although they aim at completeness, solvers nevertheless employ simple, non-relational domains. They rely on collections of simple domains (similar to disjunctive completions) to reach the desired precision. The domains are homogeneous and cannot mix variables of different type. On the contrary, AI enjoys a rich collection of abstract domains, including relational and heterogeneous ones.

On the algorithmic side, AI and CP share common ideas. Iterated propagations in CP are similar to local iterations in AI. In fact, approximating consistency in CP is similar to approximating the effect of a complex test in AI. However, search engines in CP use features, such as choice points and backtracking, that have no equivalent in AI. Dually, the widening from AI has no equivalent in CP, as CP does not employ increasing iterations but only decreasing ones.

Finally, while abstract interpreters are usually defined in a very generic way and parametrized by arbitrary abstract domains, solvers are far less flexible and embed choices of abstractions (such as domains and consistencies) as well as concrete semantics (the type of variables) in their design. In the following, we will design an abstract solver that avoids these pitfalls and can benefit from the large library of abstract domains designed for AI.

3 An Abstract Constraint Solver

We now present our main contribution: expressing constraint solving as an abstract interpreter, which involves defining concrete and abstract domains, abstract operators for split and consistency, and an iteration scheme.

3.1 Concrete Solving

A CSP is similar to the analysis of a conjunction of tests and can be formalized in terms of local iterations. We consider as concrete domain $\mathcal{D}$ the subsets of the CSP search space $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ (Def. 1), *i.e.,* $(\mathcal{P}(\hat{D}), \subseteq, \emptyset, \cup)$. Each constraint C_i corresponds to a concrete lower closure operator $\rho_i : \mathcal{P}(\hat{D}) \to \mathcal{P}(\hat{D})$, such that $\rho_i(X)$ keeps only the points in X satisfying C_i. The concrete solution of the problem is simply $S = \rho(\hat{D})$, where $\rho = \rho_1 \circ \cdots \circ \rho_p$. It is expressed in fixpoint form as $\mathrm{gfp}_{\hat{D}}\, \rho$.

3.2 Abstract Domains

Solvers do not manipulate individual points in $\hat{D}$, but rather collections of points of certain forms, such as boxes, called domains in CP. We now show that CP-domains are elements of an abstract domain $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \perp^\sharp, \sqcup^\sharp)$ in AI, which depends on the chosen consistency. In addition to standard AI operators, we require a monotonic *size function* $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ that we will use later as a termination criterion (Def. 10).

Example 1. Generalized arc-consistency (Def. 6) corresponds to the abstract domain of integer Cartesian products $\mathcal{S}^\sharp$ (Def. 2), ordered by element-wise set inclusion. It is linked with the concrete domain $\mathcal{D}$ by the standard Cartesian Galois connection:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \mathcal{S}^\sharp \\
&\gamma_a(S_1, \ldots, S_n) = S_1 \times \cdots \times S_n \\
&\alpha_a(X) = \lambda i. \{x \mid \exists (x_1, \ldots, x_n) \in X,\ x_i = x\}
\end{aligned}$$

The size function τ_a uses the size of the largest component, minus one, so that singletons have size 0:

$$\tau_a(S_1, \ldots, S_n) = \max_i(|S_i| - 1)$$

Example 2. Bound consistency (Def. 7) corresponds to the domain of integer boxes $\mathcal{I}^\sharp$ (Def. 3), ordered by element-wise interval inclusion. We have a Galois connection, and use as size function the length of the largest dimension:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} \mathcal{I}^\sharp \\
&\gamma_b([\![a_1, b_1]\!], \ldots, [\![a_n, b_n]\!]) = [\![a_1, b_1]\!] \times \cdots \times [\![a_n, b_n]\!] \\
&\alpha_b(X) = \lambda i. [\![\min \{x \in \mathbb{Z} \mid \exists (x_1, \ldots, x_n) \in X,\ x_i = x\}, \\
&\qquad\qquad\quad \max \{x \in \mathbb{Z} \mid \exists (x_1, \ldots, x_n) \in X,\ x_i = x\}]\!] \\
&\tau_b([\![a_1, b_1]\!], \ldots, [\![a_n, b_n]\!]) = \max_i(b_i - a_i)
\end{aligned}$$

Example 3. Hull consistency (Def. 8) corresponds to the domain of boxes with floating-point bounds $\mathcal{B}^\sharp$ (Def. 4). We use the following Galois connection and size function:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \mathcal{B}^\sharp \\
&\gamma_h([a_1, b_1], \ldots, [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n] \\
&\alpha_h(X) = \lambda i. [\max \{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\ x_i \geq x\}, \\
&\qquad\qquad\quad \min \{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\ x_i \leq x\}] \\
&\tau_h([a_1, b_1], \ldots, [a_n, b_n]) = \max_i(b_i - a_i)
\end{aligned}$$

We observe that to each choice corresponds a classic non-relational abstract domain, which is an homogeneous Cartesian product of identical single-variable domains. However, this needs not be the case: new solvers can be designed beyond the ones considered in traditional CP by varying the abstract domains further. A first idea is to apply different consistencies to different variables which permits, in particular, mixing variables with discrete domains and variables with continuous domains. A second idea is to parametrize the solver with other abstract domains from the AI literature, in particular relational domains, which we illustrate below.

Example 4. The octagon domain $\mathcal{O}^\sharp$ [18] assigns a (floating-point) upper bound to each binary unit expression $\pm v_i \pm v_j$ on the variables $v_1, \ldots, v_n$. It enjoys a Galois connection, and we use the size function from [20]:

$$
\begin{array}{l}
\mathcal{D} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathcal{O}^\sharp \\
\mathcal{O}^\sharp = \{\alpha v_i + \beta v_j \,|\, i, j \in [\![1, n]\!],\ \alpha, \beta \in \{-1, 1\}\,\} \to \mathbb{F} \\
\gamma_o(X^\sharp) = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \,|\, \forall i, j, \alpha, \beta,\ \alpha x_i + \beta x_j \leq X^\sharp(\alpha v_i + \beta v_j)\} \\
\alpha_o(X) = \lambda(\alpha v_i + \beta v_j).\min\{x \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X,\ \alpha x_i + \beta x_j \leq x\} \\
\tau_o(X^\sharp) = \min(\max_{i,j,\beta}(X^\sharp(v_i + \beta v_j) + X^\sharp(-v_i - \beta v_j)), \\
\qquad\qquad\qquad \max_i (X^\sharp(v_i + v_i) + X^\sharp(-v_i - v_i))/2)
\end{array}
$$

Example 5. The polyhedron domain $\mathcal{P}^\sharp$ [9] abstract sets as convex, closed polyhedra. Modern implementations [15] generally follow the "double description approach" and maintain two dual representations for each polyhedron: a set of linear constraints and a set of generators (vertices and rays, although our polyhedra never feature rays as they are bounded). There is no abstraction function α for polyhedra, and so, no Galois connection. Operators are generally easier on one representation. In particular, we define the size function on generators as the maximal Euclidian distance between pairs of vertices:

$$\tau_p(X^\sharp) = \max_{g_i, g_j \in X^\sharp} ||g_i - g_j||$$

3.3 Constraints and Consistency

We now assume that an abstract domain $\mathcal{D}^\sharp$ underlying the solver is fixed. Given the concrete semantics of the constraints $\rho = \rho_1 \circ \cdots \circ \rho_p$, and if $\mathcal{D}^\sharp$ enjoys a Galois connection $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, then the semantics of the perfect propagator achieving the consistency for all the constraints is simply: $\alpha \circ \rho \circ \gamma$. Solvers achieve this algorithmically by applying the propagator for each constraint in turn until a fixpoint is reached or, when this process is deemed too costly, return before a fixpoint is reached. By observing that each propagator corresponds to an abstract test transfer function $\rho_i^\sharp$ in $\mathcal{D}^\sharp$, we retrieve the local iterations proposed by Granger to analyze conjunctions of tests [11]. A trivial narrowing is used here: stop refining after an iteration limit is reached.

Additionally, each $\rho_i^\sharp$ can be internally implemented by local iterations [11], a technique which is used in both the AI and CP communities. A striking connection is the analysis in non-relation domains using forward-backward iterations on expression trees [17, §2.4.4], which is extremely similar to the HC4-revise algorithm [3] developed independently for CP.

When there is no Galois connections (as for polyhedra), there is no equivalent to consistency. Nevertheless, we can still use local iterations on approximate test transfer functions $\rho_i^\sharp$, which serve the same purpose: to remove some points that do not satisfy the constraints.

3.4 Disjunctive Completion and Split

In order to approximate the solution to an arbitrary precision, solvers use a coverage of finitely many abstract elements from $\mathcal{D}^\sharp$. This corresponds in AI to the notion of disjunctive completion. We now consider the abstract domain $\mathcal{E}^\sharp = \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$, and equip it with the Smyth order $\sqsubseteq^\sharp_{\mathcal{E}}$, a classic order for disjunctive completions defined as:

$$X^\sharp \sqsubseteq^\sharp_{\mathcal{E}} Y^\sharp \iff \forall B^\sharp \in X^\sharp, \exists C^\sharp \in Y^\sharp, B^\sharp \sqsubseteq^\sharp C^\sharp$$

The creation of new disjunctions is achieved by a split operation $\oplus$, that splits an abstract element into two or more elements:

Definition 9 (Split Operator). *A* split operator $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ *satisfies:*

1. $\forall e \in \mathcal{D}^\sharp$, $|\oplus(e)|$ *is finite,*
2. $\forall e \in \mathcal{D}^\sharp$, $\forall e_i \in \oplus(e)$, $e_i \sqsubseteq^\sharp e$, *and*
3. $\forall e \in \mathcal{D}^\sharp$, $\gamma(e) = \bigcup\{\gamma(e_i) \mid e_i \in \oplus(e)\}$.

Condition 2 implies $\oplus(e) \sqsubseteq^\sharp_{\mathcal{E}} \{e\}$. Condition 3 implies that $\oplus$ is an abstraction of the identity; thus, $\oplus$ can be freely applied at any place during the solving process without altering the AI-soundness (over-approximation). We now present a few example splits.

Example 6 (Split in $\mathcal{S}^\sharp$). The instantiation of a variable v_i in a discrete domain $X^\sharp = (S_1, \ldots, S_n) \in \mathcal{S}^\sharp$ is a split operator:

$$\oplus_a(X^\sharp) = \{(S_1, \ldots, S_{i-1}, \{x\}, S_{i+1}, \ldots, S_n) \mid x \in S_i\}$$

Example 7 (Split in $\mathcal{B}^\sharp$). Cutting a box in two along a variable v_i in a continuous domain $X^\sharp = (I_1, \ldots, I_n) \in \mathcal{B}^\sharp$ is a split operator:

$$\oplus_h(X^\sharp) = \{(I_1, \ldots, I_{i-1}, [a, h], I_{i+1}, \ldots, I_n),\ (I_1, \ldots, I_{i-1}, [h, b], I_{i+1}, \ldots, I_n)\}$$

where $I_i = [a, b]$ and $h = (a + b)/2$ rounded in $\mathbb{F}$ in any direction.

Example 8 (Split in $\mathcal{O}^\sharp$). Given a binary unit expression $\alpha v_i + \beta v_j$, we define the split on an octagon $X^\sharp \in \mathcal{O}^\sharp$ along this expression as:

$$\oplus_o(X^\sharp) = \{X^\sharp[(\alpha v_i + \beta v_j) \mapsto h],\ X^\sharp[(-\alpha v_i - \beta v_j) \mapsto -h]\}$$

where $h = (X^\sharp(\alpha v_i + \beta v_j) - X^\sharp(-\alpha v_i - \beta v_j))/2$, rounded in $\mathbb{F}$ in any direction.

Example 9 (Split in $\mathcal{P}^\sharp$). Given a polyhedron $X^\sharp \in \mathcal{P}^\sharp$ represented as a set of linear constraints, and a linear expression $\sum_i \beta_i v_i$, we define the split:

$$\oplus_p(X^\sharp) = \{X^\sharp \cup \{\textstyle\sum_i \beta_i v_i \leq h\}, X^\sharp \cup \{\textstyle\sum_i \beta_i v_i \geq h\}\}$$

where $h = (\min_{\gamma(X^\sharp)} \sum_i \beta_i v_i + \max_{\gamma(X^\sharp)} \sum_i \beta_i v_i)/2$ can be computed by the Simplex algorithm.

These splits are parametrized by the choice of a direction of cut (some variable or expression). For non-relational domains we can use two classic strategies from CP: split each variable in turn, or split along a variable with maximal size (*i.e.,* $|S_i|$ or $b_i - a_i$). These strategies lift naturally to octagons by replacing the set of variables with the (finite) set of unit binary expressions (see also [20]). For polyhedra, one can bisect the segment between two vertices that are the farthest apart, in order to minimize τ_p. However, even for relational domains, we can use a faster and simpler non-relational split, *e.g.,* cut along the variable with the largest range.

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:

Definition 10. *The operators $\tau : \mathcal{D}^\sharp \to \mathcal{R}^+$ and $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ are compatible if, for any reductive operator $\rho^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ (i.e., $\forall X^\sharp \in \mathcal{D}^\sharp, \rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$) and any family of choice operators $\pi_i : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ (i.e., $\forall Y^\sharp \in \mathcal{E}^\sharp, \pi_i(Y^\sharp) \in Y^\sharp$), we have:*

$$\forall e \in \mathcal{D}^\sharp,\ \forall r \in \mathbb{R}^{>0}, \exists K \ s.t.\ \forall j \geq K,\ (\tau \circ \pi_j \circ \oplus \circ \rho \circ \cdots \circ \pi_1 \circ \oplus \circ \rho)(e) \leq r$$

Each of the split function we presented above, $\oplus_a$, $\oplus_h$, $\oplus_o$, and $\oplus_p$, is compatible with the size function τ_a, τ_h, τ_o, and τ_p we proposed on the corresponding domain.

The search procedure can be represented as a search tree where each node corresponds to a search space and the children of a node are constructed by applying the split operator on the parent and then applying a reduction. With this representation, the set of nodes at a given depth corresponds to a disjunction over-approximating the solution. Moreover, a series of reduction (ρ), selection (π), and split ($\oplus$) operators corresponds to a tree branch. Definition 10 states that each branch of the search tree is finite.

3.5 Abstract Solving

We are now ready to present our solving algorithm, in Fig. 2. It maintains in toExplore and sols two disjunctions in $\mathcal{E}^\sharp$, and iterates the following steps: choose an abstract element e from toExplore (**pop**), apply the consistency ($\rho^\sharp$), and either discard the result, add it to the set of solutions sols, or split it ($\oplus$). The solver starts with the maximal element $\top^\sharp$ of $\mathcal{D}^\sharp$, which represents $\gamma(\top^\sharp) = \hat{D}$.

Correctness. At each step, $\bigcup\{\gamma(x) \mid x \in \text{toExplore} \cup \text{sols}\}$ is an over-approximation of the set of solutions, because the consistency $\rho^\sharp$ is an abstraction of the concrete semantics ρ of the constraints and the split $\oplus$ is an abstraction of the identity. We note that abstract elements in sols are consistent and either contain only solutions or are smaller than r. The algorithm terminates when toExplore is empty, at which point sols over-approximates the set of solutions with consistent elements that contain only solutions or are smaller than r. To compute the exact set of solutions in the discrete case, it is sufficient to choose $r < 1$.

The termination is ensured by the following proposition:

```
list of abstract domains sols ← $\emptyset$                /*stores the abstract solutions*/
queue of abstract domains toExplore ← $\emptyset$ /*stores the abstract elements to explore*/
push $\top^\sharp$ in toExplore /*initialization with the abstract search space: $\gamma(\top^\sharp) = \hat{D}$*/

while toExplore $\neq \emptyset$ do
    $e \leftarrow$ pop($toExplore$)
    $e \leftarrow \rho^\sharp(e)$
    if $e \neq \emptyset$ then
        if $\tau(e) \leq r$ or isSol($e$)  /*isSol(e) returns true if e contains only solutions*/
        then
            sols ← sols $\cup\ e$
        else
            push $\oplus(e)$ in toExplore
```

Fig. 2. Our generic abstract solver.

Proposition 1. *If τ and $\oplus$ are compatible, the algorithm in Fig. 2 terminates.*

Proof. The search tree is finite. Otherwise, as its width is finite by Def. 9, there would exist an infinite branch (König's lemma), which would contradict Def. 10.

The solver in Fig. 2 uses a queue data-structure, and splits the oldest abstract element first. More clever choosing strategies are possible (*e.g.,* split the largest element for τ). The algorithm remains correct and terminates for any strategy.

Comparison with Abstract Interpretation. Similarly to local iterations in AI, our solver performs decreasing abstract iterations. Indeed, toExplore $\cup$ sols is decreasing for $\sqsubseteq^\sharp_\mathcal{E}$ in the disjunctive completion domain $\mathcal{E}^\sharp$ at each iteration of the loop (indeed, $\oplus(e) \sqsubseteq^\sharp_\mathcal{E} \{e\}$ and we can assume that $\rho^\sharp$ is reductive in $\mathcal{D}^\sharp$ without loss of generality). However, it differs from classic AI in two ways. Firstly, there is no split operator in AI: new components in a disjunctive completion are generally added only at control-flow joins (by delaying the abstract join $\sqcup^\sharp$ in $\mathcal{D}^\sharp$). Secondly, the solving iteration strategy is far more elaborated than in AI. The use of a narrowing is replaced with a data-structure that maintains an ordered list of abstract elements and a splitting strategy that performs a refinement process and ensures its termination. Actually, more complex strategies than the simple one we presented here exist in the CP literature. One example is the AC-5 algorithm [26] where, each time the domain of a variable changes, the variable decides which constraints need to be propagated. The design of efficient propagation algorithms is an active research area [22].

4 Experiments

We have implemented a prototype solver to demonstrate the feasibility of our approach. We describe its main features and present experimental results.

4.1 Implementation

Our prototype solver, called Absolute, is implemented in OCaml. It uses Apron, a library of numeric abstract domains intended primarily for static analysis [15]. We benefit from Apron domains (intervals, octagons, and polyhedra), its ability to hide their internal algorithms under a uniform API, and its handling of integer and real variables and of non-linear constraints.

Consistency. Apron provides a language of constraints sufficient to express many CSPs: equalities and inequalities over numeric expressions (including operators such as $+$, $-$, $\times$, $/$, $\sqrt{}$, power, modulo, and rounding to integers). The test transfer function naturally provides propagators for these constraints. Internally, each domain implements its own algorithm to handle tests, including sophisticated methods to handle non-linear constraints (such as HC4 and linearization [17]). Our solver then performs local iterations until either a fixpoint or a maximum number of iterations is reached (which is set to 3 to ensure a fast solving). In CP solvers, only the constraints containing at least one variable that has been modified during the previous step are propagated. However, for simplicity, our solver propagates all the constraints at each step of the local iteration.

Split. Currently, our solver only splits along a single variable at a time, cutting its range in two, even for relational domains and integer variables. It chooses the variable with the largest range. It uses a queue to maintain the set of abstract elements to explore (as in Fig. 2). Compared to most CP solvers, this splitting strategy is very basic. It will be improved in the future by integrating more clever strategies from the CP literature.

4.2 Exemple of AI-solving with Absolute

In order to make the abstract solving process clear, we detail here an example with a very simple problem. Consider a CSP on two continuous variables v_1 and v_2 taking their values in $D_1 = D_2 = [-5, 5]$, and the constraints $C_1 : x^2 + y^2 \leq 4$ and $C_2 : (x - 2)^2 + (y + 1)^2 \leq 1$.

Figure 3 shows the first iterations of the AI-solving method for this CSP. The root corresponds to the initial search space after applying the reduction based on HC4 as explained above ($D_1 = [1, 2], D_2 = [-1.73, 0]$). Its successor nodes correspond to the search spaces obtained after splitting the domain D_2 in half and applying the reduction to the new states. These two steps (split and reduction) are repeatedly applied until all the solutions have been found, but Figure 3 only shows the first three steps.

At a given depth in the search tree, the current approximation of the solution space is made of the disjunction of the abstract elements currently investigated. Figure 4 shows these disjunctions for the search tree depicted in Fig. 3.

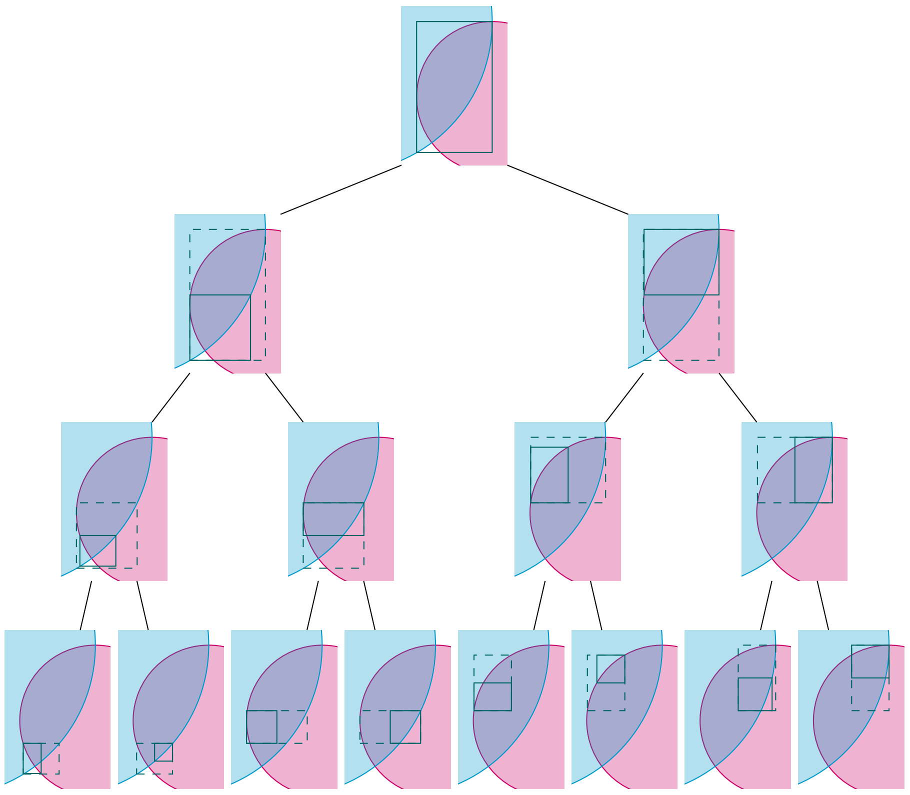
Fig. 3. First iterations of the AI-solving method.

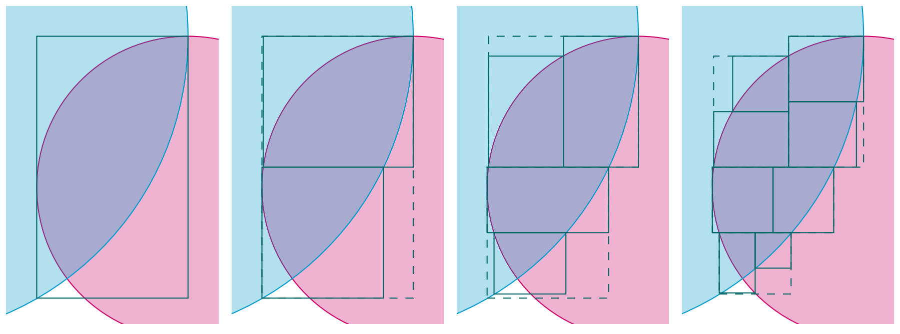

Fig. 4. Disjunctions of the first iterations of the AI-solving method search tree given in Fig. 3.

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	$=$	0.009	0.018	0.053	0.048
nbody5.1	6	$=$	32.85	708.47	0.027	$\geq$ 1h
ipp	8	$=$	0.66	9.64	19.28	1.46
brent-10	10	$=$	7.96	4.57	0.617	$\geq$ 1h
KinematicPair	2	$\leq$	0.013	0.018	0.016	0.011
biggsc4	4	$\leq$	0.011	0.022	0.096	0.029
o32	5	$\leq$	0.045	0.156	0.021	0.263

Table 1. CPU time in seconds to find the first solution with Ibex and Absolute.

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	$=$	0.02	0.10	0.26	0.14
nbody5.1	6	$=$	95.99	1538.25	27.08	$\geq$ 1h
ipp	8	$=$	38.83	39.24	279.36	817.86
brent-10	10	$=$	21.58	263.86	330.73	$\geq$ 1h
KinematicPair	2	$\leq$	59.04	23.14	60.78	31.11
biggsc4	4	$\leq$	800.91	414.94	1772.52	688.56
o32	5	$\leq$	27.36	22.66	40.74	33.17

Table 2. CPU time in seconds to find all solutions with Ibex and Absolute.

4.3 Experimental Results

We have run Absolute on two classes of problems: firstly, on continuous problems to compare its efficiency with state-of-the-art CP solvers; secondly, on mixed problems, that these CP solvers cannot handle while our abstract solver can.

Continuous solving. We use problems from the COCONUT benchmark[2], a standard CP benchmark with only real variables. We compare Absolute with the standard (interval-based) Ibex CP continuous solver[3]. Notice that the COCONUT problems have a relatively small number of variables, compared for instance to the number of variables that can be analyzed for a single program in AI. The difficulty of the benchmark is here due to both the expressions of the constraints (non linear with multiple variable occurences) and the high precision that is required.

Additionally, we compare Absolute to our extension of Ibex to octagons from previous work [20], which allows comparing the choice of domain (intervals versus octagons) independently from the choice of solver algorithm (classic CP solver versus our AI-based solver). Tables 1 and 2 show the run time in seconds to find all the solutions or only the first solution of each problem. Tables 3 and 4 show

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

[3] Available at `http://www.emn.fr/z-info/ibex/`.

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	$=$	145	28	45	207
nbody5.1	6	$=$	262 659	2 765 630	105	-
ipp	8	$=$	4 039	25 389	899	3 421
brent-10	10	$=$	101 701	12 744	2 113	-
KinematicPair	2	$\leq$	43	55	39	55
biggsc4	4	$\leq$	98	96	94	84
o32	5	$\leq$	87	344	85	942

Table 3. Number of nodes created to find the first solution with Ibex and Absolute.

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	$=$	551	577	147	1057
nbody5.1	6	$=$	598 521	5 536 283	7 925	-
ipp	8	$=$	237 445	99 179	39 135	2 884 925
brent-10	10	$=$	211 885	926 587	5 527	-
KinematicPair	2	$\leq$	847 643	215 465	520 847	215 465
biggsc4	4	$\leq$	3 824 249	6 038 844	2 411 741	6 037 260
o32	5	$\leq$	161 549	120 842	84 549	111 194

Table 4. Number of nodes created to find all solutions with Ibex and Absolute.

the number of nodes created to find all the solutions or only the first solution of each problem.

On average, Absolute is competitive with the traditional CP approach. More precisely, it is globally slower on problems with equalities, and faster on problems with inequalities. This difference of performance seems to be related to the following ratio: the number of constraints in which a variable appears over the total number of constraints. As said previously, at each iteration, all the constraints are propagated even those for which none of their variables have changed. This increases the computation time at each step and thus increases the overall time. For instance, in the problem `brent-10`, there are ten variables, ten constraints, and each variable appears in at most three constraints. If only one variable has been modified, we will nevertheless propagate all ten constraints, instead of three at most. This may explain the timeouts observed on problems `brent-10` and `nbody5.1` with Absolute.

Moreover, in our solver, the consistency loop is stopped after three iterations while, in the classic CP approach, the fixpoint is reached. The consistency in Absolute may be less precise than the one used in Ibex, which reduces the time spent during the propagation step but may increase the search phase. This probably explains why in tables 3 and 4 the number of nodes created during the solving process with Absolute is most of the times larger than the one with Ibex. Less reductions are performed thus more splitting operations are needed, hence more nodes are created during the solving process.

name	# vars		ctr type	First Solution			All solutions		
	int	real		$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$	$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
gear4	4	2	$=$	0.016	0.036	0.296	0.017	0.048	0.415
st_miqp5	2	5	$\leq$	0.672	1.152	$\geq$ 1h	2.636	3.636	$\geq$ 1h
ex1263	72	20	$=$ $\leq$	8.747	$\geq$ 1h	$\geq$ 1h	473.933	$\geq$ 1h	$\geq$ 1h
antennes_4_3	6	2	$\leq$	3.297	22.545	$\geq$ 1h	520.766	1562.335	$\geq$ 1h

Table 5. CPU time, in seconds, to solve mixed problems with Absolute.

name	# vars		ctr type	First Solution			All solutions		
	int	real		$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$	$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
gear4	4	2	$=$	43	226	226	67	501	501
st_miqp5	2	5	$\leq$	2 247	2 247	-	7 621	7 621	-
ex1263	72	20	$=$ $\leq$	8544	-	-	493 417	-	-
antennes_4_3	6	2	$\leq$	17 625	40 861	-	2 959 255	6 657 237	-

Table 6. Number of nodes created to solve mixed problems with Absolute.

These experimentations show that our prototype, which only features quite naïve CP strategies, behaves reasonably well on a classic benchmark. Further studies will include a deeper analysis of the performances and improvements of Absolute on its identified weaknesses (splitting strategy, propagation loop).

Mixed discrete-continuous solving. As CP solvers seldom handle mixed problems, no standard benchmark exists. We thus gathered problems from MinLPLib,[4] a library of mixed optimisation problems from the Operational Research community. These problems are not satisfaction CSPs, but optimization problems, with constraints to satisfy and a function to minimize. We thus needed to turn them info satisfaction CSPs. Following the approach in [4], we replaced each optimization criterion $\min f(x)$ with a constraint $|f(x) - \text{best\_known\_value}| \leq \epsilon$. We compared Absolute to the mixed solving scheme from [4], using the same ϵ and benchmarks, and found that they have similar run times (we do not provide a more detailed comparison as it would be meaningless due to the machine differences).

More interestingly, we observe that Absolute can solve mixed problems in reasonable time and behaves better with intervals than with relational domains. A possible reason is that current propagations and heuristics are not able to fully use relational information available in octagons or polyhedra. Previous works [20] suggest that a carefully designed split is key to efficient octagons; future work will incorporate ideas from [20] into our solver and develop them further. Already, Absolute is able to naturally cope with mixed CP problems

[4] Available at `http://www.gamsworld.org/minlp/minlplib.htm`.

in a reasonable time, opening the way to new CP applications such as robotic localization [14] or geometric problems [1].

5 Conclusion

In this paper, we have exposed some links between AI and CP, and used them to design a CP solving scheme built entirely on abstract domains. The preliminary results obtained with our prototype are encouraging and open the way to the development of hybrid CP–AI solvers able to naturally handle mixed constraint problems. In future work, we wish to improve our solver by adapting and integrating advanced methods from the CP literature. The areas of improvement include: split operators for abstract domains, specialized propagators (such as octagonal consistency or global constraints), and improvements to the propagation loop. We built our solver on abstractions in a modular way, so that existing and new methods can be combined together, as is the case for reduced products in AI. Ultimately, each problem should be automatically solved in the abstract domains which best fit it, as it is the case in AI. A natural future work is thus the development of new abstract domains adapted to specific constraint kinds. Another exciting development would be to use some methods form CP in an AI-based static analyzer. Areas of interest include: decreasing iteration methods, which are more advanced in CP than in AI, the use of a split operator in disjunctive completion domains, and the ability of CP to refine an abstract element to achieve completeness. Finally, it also remains to understand how fixpoint extrapolation operators, such as widenings, which are very popular in AI, can be exploited in CP solvers.

References

1. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proc. of the 13th Int. Conf. on Principles and Practice of Constraint Programming*, 2007.
2. F. Benhamou. Heterogeneous constraint solvings. In *Proc. of the 5th Int. Conf. on Algebraic and Logic Programming*, pages 62–76, 1996.
3. F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revisiting hull and box consistency. In *Proc. of the 16th Int. Conf. on Logic Programming*, pages 230–244, 1999.
4. N. Berger and L. Granvilliers. Some interval approximation techniques for MINLP. In *SARA*, 2009.
5. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*. AIAA, 2010.
6. G. Chabert, L. Jaulin, and X. Lorca. A constraint on the number of distinct vectors with application to localization. In *CP*, pages 196–210, 2009.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf.*

Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252. ACM Press, 1977.

8. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96, 1978.
10. V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analysers. In *Proc. of the 19th Int. Static Analysis Symposium (SAS'12)*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.
11. P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Proc. of the 12th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1992.
12. N. Halbwachs and J. Henry. When the decreasing sequence fails. In *Proc. of the 19th Int. Static Analysis Symposium (SAS'12)*, volume 7460 of *LNCS*, pages 198–213. Springer, 2012.
13. A. Hervieu, B. Baudry, and A. Gotlieb. Pacogen: Automatic generation of pairwise test configurations from feature models. In *Proc. of the 22nd Int. Symposium on Software Reliability Engineering*, pages 120–129, 2011.
14. L. Jaulin and S. Bazeille. Image shape extraction using interval methods. In *Proc. of the 15th IFAC Symposium on System Identification*, 2009.
15. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. Computer Aided Verification (CAV 2009)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.
16. N. Lazaar, A. Gotlieb, and Y. Lebbah. A CP framework for testing CP. *Constraints*, 17(2):123–147, 2012.
17. A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Palaiseau, France, December 2004.
18. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
19. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.
20. M. Pelleau, C. Truchet, and F. Benhamou. Octagonal domains for continuous constraints. In *Proc. of the 17th Int. Conf. on Principles and Practice of Constraint Programming*, 2011.
21. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier, 2006.
22. C. Schulte and G. Tack. Implementing efficient propagation control. In *Proc. of the 3rd workshop on Techniques for Implementing Constraint Programming Systems*, 2001.
23. Choco Team. Choco: an open source Java constraint programming library. Research report 10-02-INFO, École des Mines de Nantes, 2010.
24. A. Thakur and T. Reps. A generalization of støAlmarck's method. In *Proc. of the 19th Int. Static Analysis Symposium (SAS'12)*, volume 7460 of *LNCS*, pages 334–351. Springer, 2012.
25. C. Truchet, M. Pelleau, and F. Benhamou. Abstract domains for constraint programming, with the example of octagons. *Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 72–79, 2010.
26. P. van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57, 1992.