EasyChair Preprint

№ 1906

Combination of Boxes and PolyhedraAbstractions for Constraint Solving

Ghiles Ziat, Alexandre Maréchal, Marie Pelleau, Antoine Miné and Charlotte Truchet

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

November 8, 2019

Combination of Boxes and Polyhedra Abstractions for Constraint Solving

Ghiles Ziat[1], Alexandre Maréchal[1,2], Marie Pelleau[3], Antoine Miné[1], and Charlotte Truchet[4]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[2] Univ. Grenoble Alpes, CNRS, VERIMAG, F-38000 Grenoble, France
[3] Université Côte D'Azur, CNRS, I3S, F-06100 Nice, France
[4] LS2N, UMR 6004, Université de Nantes, F-44300 Nantes, France

Abstract. This paper investigates the use of abstract domains from Abstract Interpretation (AI) in the field of Constraint Programming (CP). CP solvers are generally very efficient on a specific constraint language, but can hardly be extended to tackle more general languages, both in terms of variable representation (discrete or continuous) and constraint type (global, arithmetic, etc.). For instance, linear constraints are usually solved with linear programming techniques, but non-linear ones have to be either linearized, reformulated or sent to an external solver. We approach this problem by adapting to CP a popular domain construction used to combine the power of several analyses in AI: the *reduced product*. We apply this product on two well-known abstract domains, Boxes and Polyhedra, that we lift to constraint solving. Finally we define general metrics for the quality of the solver results, and present a benchmark accordingly. Experiments show promising results and good performances.

1 Introduction

Constraint programming (CP) is a paradigm of declarative programming, in which problems are described in mathematical terms involving constraints (*i.e.* first-order logic formulas) over variables, and then solved using a constraint solver. Solvers often focus on a few *constraint languages*, which are families of constraints such as linear (in)equalities over the reals, over the integers, polynomial constraints, real constraints with mathematical functions (sin, cos, log...), integer cardinality constraints, etc. Yet, most solvers share two common ingredients: first, they use propagation algorithms to reduce the search space by reasoning on the constraints, without losing solutions. Second, they usually feature a branching process that consists in adding hypotheses (e.g. variable instantiation or domain splitting) to the problem. If the new problem is proved infeasible, the solver backtracks on the current hypotheses and makes new ones to explore other parts of the search space. In classic CP solvers, the variables are always considered independent, the only relations between them being the constraints. In practice, solvers thus work in the Cartesian product of the variables domains, which can be real intervals with floating point bounds, integer intervals or finite

integer sets. Each of these domain representations comes with specific propagators, such as Hull consistency [3] for real domains or bound-consistency [23, 18] for integer intervals, to name a few.

The notion of abstract domain has been introduced for over-approximating sets of interests (traces of programs) in order to capture specific properties among them. Previous works [17] showed how to unify the core constraint solving methods by re-defining a generic notion of abstract domain, augmented with CP-oriented operations.

A generic solving process based on abstract domains has been introduced in [16], and implemented in the AbSolute abstract solver. Given an abstract domain (e.g. the Cartesian products of real intervals, also called *boxes*), the solver combines propagators and branching operators well defined for this abstract domain (for instance on boxes: Hull consistency propagation and interval splitting). An important feature of this solving method is its modularity: the same formal method can be parametrized with different abstract domains. The main properties of the solver, which are termination, completeness, and soundness, depend on the properties of the abstract domain it uses.[5]

In this fashion, a solver can benefit from the many abstract domains that have already been defined in Abstract Interpretation (AI) to tackle specific program properties: Intervals [6], Polyhedra [9], Octagons [14, 15], etc. For instance, Octagons have been adapted to continuous constraint solving with *ad-hoc* propagation and exploration heuristics [17]. Abstract domains feature different precision and come at different costs: for instance, Octagons are costlier than Intervals but more precise. Also, some domains such as Ellipsoids are designed to capture very specific properties and ignore other ones, or propose very coarse approximations for them. Choosing which domain to use is not a trivial task as these facts must be taken into account.

In addition to abstract domains, AI defines a set of abstract domain transformers, building upon one or several abstract domains to improve or combine their precision. Such transformers are very useful in practice as they create more expressive combined domains in a modular and generic way. For example the Trace Partitioning transformer [19] partitions execution traces of a program according to the control flow (*e.g.,* which branch of a conditional statement was taken), leading to a path sensitive analysis. It focuses on the abstraction of the control flow and delegates the value analysis to another domain (generally a numeric one), whose accuracy will benefit from the partitioning. In the following, we will extensively use the Reduced Product [8], another very popular domain transformer. A reduced product combines two domains to represent conjunctions of properties from both of them. Additionally, operations in the reduced product apply a reduction operator to communicate information between the base domains, thus improving the precision.

[5] Contrarily to AI, in CP the term completeness refers to an algorithm which does not lose solutions (over-approximating the solution set), while soundness means that the solver under-approximates the solution set. This vocabulary can be misleading. In the following, we will use "over/under-approximation" to avoid any ambiguity.

Contribution. In this article, we first present a Constraint Programming version of the Box and Polyhedra abstract domains [9]. We then introduce a version of the Reduced Product domain transformer adapted to CP purposes, and we detail a constraint attribution operators for the Box-Polyhedra reduced product. Finally, we present an implementation in the AbSolute solver and experiments made on the Coconut benchmark [22].

To be usable in a CP framework, we will have to define on each abstract domain (1) a *split* operator, to implement the branching process; (2) a *size* function, to determine when the solver finishes; (3) and *propagators* for given constraint languages. Compared to the classic reduced product of AI, our version introduces a hierarchy between the domains, one of them being specialized to a certain kind of problems only, in order to avoid a redundancy of information between the two components of the product.

This paper is organized as follows: Section 2 recalls the definitions and results on abstract domains needed afterwards, in particular the generic notion of abstract domain for CP. Section 3 introduces the Box and Polyhedra abstract domains. Section 4 then explains how to build Reduced Products domains for CP, and details the Box-Polyhedra reduced product. Section 5 presents experiments using AbSolute with the new Box-Polyhedra domain. Finally, Section 6 concludes.

Related works. The links between CP and AI have already been highlighted in previous works. The seminal work of Apt [1] expresses the propagation loop which consists in propagating the unfeasible set of some constraints as chaotic iterations in a well-chosen lattice. In the same spirit, [2] defines propagators, whether discrete or continuous, in a similar framework. Later, [20] keeps the same idea and weakens the conditions on the propagators while keeping the convergence of the propagation loop. All these works focus on propagation. We go one step further by expressing the splitting and size operators in the same framework, thus taking into account the whole solving process.

A work more related to ours is [21], which also investigates the use of abstract domains in CP and also mainly focuses on propagation. The key difference is the way we build the abstract domain. [21] defines abstractions solely through Galois connections, which is an important restriction as it bans interesting abstractions such as polyhedra or zonotopes [11]. On the contrary, we can support a larger set of abstractions, including those for which there is no Galois connection.

2 Abstract Constraint Solving

This section introduces the core notions of CP solving, and the extension of abstract domains to the CP context.

2.1 Constraint Programming Background

Constraint solvers can deal with problems written as Constraint Satisfaction Problems (CSP), where *variables* represent the unknowns of the problem. Each

variable takes its value from a given set, usually called *domain*, and constraints express the relations between the variables. Note that the domains defined in CP are not abstract domains as defined in AI. Next section will clarify this.

Definition 1. *A Constraint Satisfaction Problem is a triplet $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, where n and m are respectively the number of variables and the number of constraints of the problem:*

- $\mathcal{V} = \{v_1, ..., v_n\}$ *are variables representing the unknowns of the problem,*
- $\mathcal{D} = \{d_1, ..., d_n\}$ *are domains for the variable, such that $v_k \in d_k, \forall k \in [1, n]$,*
- $\mathcal{C} = \{c_1, ..., c_m\}$ *are constraints over the variables.*

Constraints of a CSP are defined in a given constraint language, *i.e.* a family of first-order logical formulæ. For simplicity, we focus in this section on the case of real variables, where the domains are real intervals with floating-point bounds, and the constraints can be written using arithmetic operators, common mathematical functions (sin, cos, log, and any function which can be computed on intervals), and a relation operator within $\{=, \neq, <, \leq\}$. The corresponding abstract domain will be formally defined in Section 3.1. Many constraint languages exist in the literature, in particular on finite domains.

We call an *instance* a total mapping $\mathcal{V} \rightarrow \mathcal{D}$ from variables to their domain. A solution of a CSP is an instance that satisfies all of the constraints. If the solutions are not computer-representable, as it is the case with variables taking real values, then solving the CSP means finding domains for the variables which are either entirely solutions (all the instances inside the domains satisfy the constraints), or are smaller than a given precision (on interval domains for instance, the size of the domain is the interval length).

The search space is usually either too large (in the discrete case, its size is exponential in the number of variables) or infinite (in the continuous case, solutions for real variable may not be computer-representable) to be explored exhaustively. A key ingredient of constraint solving is the notion of propagation, which relies on the constraints to reduce the search space.

Definition 2. *Let $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP, and let $c \in \mathcal{C}$ be a constraint. A propagator ρ for c is a function from $\mathscr{P}(\mathcal{D})$ to itself such that:*
- $\forall \mathcal{D}' \in \mathscr{P}(\mathcal{D}), \rho(\mathcal{D}') \subseteq \mathcal{D}'$,
- $\forall \mathcal{D}' \in \mathscr{P}(\mathcal{D}), \forall (x_1, \ldots, x_n) \in \mathcal{D}', c(x_1, \ldots, x_n) \implies (x_1, \ldots, x_n) \in \rho(\mathcal{D}')$

The first condition makes the propagators always decreasing for the $\subset$-order.[6] The second condition ensures that a propagator does not remove solutions for its constraint. Constraint propagators are usually built to tighten the search space as much as possible. For instance, Hull-consistency propagation on boxes [3], which is similar to the bottom-up top-down algorithm in AI, reduces the domains by analyzing the expressions inside the constraints with interval arithmetics.

[6] Contrarily to the evolution of abstract elements during a static analysis in AI, constraint domains always decrease.

The propagation step in a solver applies the propagators for each constraint, until it reaches a given consistency. Consistencies are properties on the satisfiability of a problem. The application of a propagator makes it possible to establish such properties. For example, the HC4 propagator [3] establishes the Hull-Consistency property (*i.e.* it computes the smallest box containing all of the solutions of the problem). For example, given two real variables x and y defined respectively on the intervals $[0; 5]$ and $[0; 10]$, the smallest box that contains all of the solutions of the constraint $x^2 + y^2 \leq 4$ is the store that maps x to $[0; 2]$ and y to $[0; 2]$. Propagation is usually not sufficient to find solutions of the problems, for instance when the solution set cannot be exactly represented by a single abstract element (*e.g.,* Cartesian products cannot represent complex shapes). Thus, the solver alternates propagation steps and choice operations (split), as detailed below.

2.2 Abstract Domains for Constraint Solving

A key point in our work is the use and combination of abstract domains. In AI, they have been introduced to over-approximate program states [6]. 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 *box*. An abstract domain is a partially ordered set (a *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.

Abstract domains have already been extended to be used in a CP solver in [16]. We recall the main definitions and algorithms in this section. A classic CP solver alternates two main steps: propagation and search. The abstract-solving method is defined by lifting up these operations to abstract domains. An abstract CP domain must thus feature a propagation operation, a size function and a split operator. Propagation is quite specific in our work, and is defined in the next subsection.

Definition 3 (Abstract Domain for Constraints). *An abstract domain is given by:*
- *a poset* $(E, \subseteq)$*, with a computer representation for the elements of* E*,*
- *a propagator* $\rho : E \to E$ *for each constraint* c*,*
- *a splitting operator on* E*,* $\oplus_E : E \to \mathscr{P}(E)$*,*
- *a size function* $\tau_E : E \to \mathbb{R}^+$*.*

Here, the poset $(E, \subseteq)$ defines the sets of points that can be exactly represented (boxes, octagons, etc). The propagator must return an over-approximation, so that it does not lose solutions (as in Def. 2). The size function 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 *too small to be split* if $\tau_E(e)$ is less than or equal to a parameter $r \in \mathbb{R}^+$. Moreover, if an element e is an atom of E, $\tau_E(e)$ should be equal to 0 as it is not possible to split e into smaller elements (*e.g.,* interval singletons).

Algorithm 1: Solving with abstract domains.

```
function solve(e, C)              // e: initial abstract element, C: constraints
    sols ← ∅                                          // abstract solutions
    toExplore ← ∅                           // abstract elements to explore
    push e in toExplore
    while toExplore ≠ ∅ do
        e ← pop(toExplore)
        e ← ρ(e, C)                      // propagation of all constraints
        if e ≠ ⊥ then
            if τ_E(e) ≤ r or ∀c ∈ C, c(e) then
                sols ← sols ∪ e
            else
                ∀e_i ∈ ⊕_E(e), push e_i in toExplore              // splitting
```

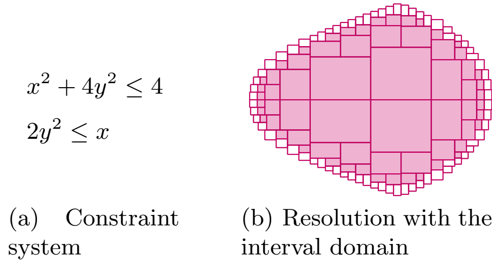
We call split the action of dividing an abstract element into smaller ones w.r.t. $\subseteq$. The split operator $\oplus_E$ must respect some conditions and should be designed in accordance with τ_E.

Definition 4. *Let $(E, \subseteq)$ be a poset. A split operator $\oplus_E : E \to \mathscr{P}(E)$ is such that, for $e \in E$ an abstract element, we have:*

- $\cup \oplus_E (e) = e$ *(no solution must be lost, nor added),*
- $| \oplus_E (e)|$ *is finite (this ensures finite width of the search tree),*
- $\exists \epsilon > 0, \forall e, \forall e_i \in \oplus_E(e), \tau_E(e_i) \leq \tau_E(e) - \epsilon$*, (this ensures finite depth of the search tree, hence termination).*

If an abstract domain features all these operators, it can be used in the abstract solving method defined in [16], which solves CSPs by computing and refining covers of their solution space using abstract elements. Algorithm 1 gives its pseudo-code. It proceeds as follows: given an initial abstract element e, supposed to represent exactly the domains of definitions of the variables, and a set of constraints $\mathcal{C}$, we maintain a list of abstract elements `toExplore`, containing all the abstract elements which remain to be explored, and initialized with e. The main loop takes one element in `toExplore`, and performs the propagation of the constraints on e. If e is empty, then it contains no solution and is discarded. Otherwise, if e either fully satisfies all the constraints, or is small enough, it is added to the solutions of the problem. And in the other case, the status of e remains undecided, thus e is split and the resulting new elements are added to `toExplore`. Any abstract domain can be used within this algorithm, provided that the constraint propagators are well defined.

Figure **??** shows the result of this solving method on an example from the Coconut benchmark. We can distinguish two kinds of elements in the resulting cover: the one that are proven to satisfy the CSP, and the one that were too small to be split. Considering only the former gives an under-approximation of the



(a) Constraint system

(b) Resolution with the interval domain

Fig. 1. Resolution of a continuous constraint problem using the interval abstract domain.

solution set and considering both kinds gives the same result as the continuous solving method in CP, that is a union of boxes over-approximating the solutions.

3 Boxes and Polyhedra as Constraint Abstract Domains

As explained above, the notion of abstract domain coming from AI can be adapted to be used in CP. In this Section, we introduce two numerical abstract domains, Boxes and Polyhedra, in their CP version.

3.1 A Non-Relational Abstract Domain: Continuous Boxes

In non-relational domains, variables are analyzed independently. In other words, each variable is assigned to a domain regardless of the domains of the other variables. We detail here the case of boxes, where a variable domain is a real interval with floating-point bounds.

In classic CP solvers over continuous variables, the domains are usually represented as intervals with floating-point bounds. This representation is already a lattice [2]. Consider $\mathbb{F}$ the set of (non special) floating point numbers according to the IEEE norm [10]. For $a, b \in \mathbb{F}$, let $[a, b] = \{x \in \mathbb{R}, a \leq x \leq b\}$ the real interval delimited by a and b, and $\mathbb{I} = \{[a, b], a, b \in \mathbb{F}\}$ the set of all such intervals. For any interval I, we write $\underline{I}$ (resp. $\overline{I}$) its lower (resp. upper) bound. Similarly, for any real point x, $\underline{x}$ (resp. $\overline{x}$) is the floating-point lower (resp. upper) approximation of x.

Let $v_1, \ldots, v_n$ be variables over finite continuous domains $d_1, \ldots, d_n$. We call box a Cartesian product of intervals in $d_1 \times \cdots \times d_n$. Boxes built upon $\mathcal{D}$ (the initial domain of the variable) form a finite lattice:

$$\mathbb{B}(\mathcal{D}) = \left\{ \prod_i [a_i, b_i] \,|\, \forall i,\ [a_i, b_i] \subseteq d_i \right\}$$

The abstract domain is based on the lattice $\mathbb{B}$ ordered by inclusion. Its consistency is Hull-consistency [4]. The splitting operator first uses a variable selection strategy (*e.g.,* the variable with the biggest range) and then splits the domain

of the chosen variable in two. Let v_i be the variable chosen for the split and $d_i = [a_i, b_i] \in \mathbb{I}$ its domain. Let $h = \frac{a_i+b_i}{2}$ rounded to the nearest float. The split operator is:

$$\oplus_{\mathbb{B}}(d_1 \times \dots \times d_n) = \{ d_1 \times \dots \times [a_i, h] \times \dots \times d_n, \\ d_1 \times \dots \times [h, b_i] \times \dots \times d_n \}$$

The size function corresponds to the Manhattan distance between two extremities of a diagonal of a box:

$$\tau_{\mathbb{B}}([a_1, b_1] \times \ldots \times [a_n, b_n]) = \sum_i (b_i - a_i)$$

Here, $\oplus_{\mathbb{B}}$ and $\tau_{\mathbb{B}}$ are designed in accordance with Def. 4, and we have $\forall e' \in \oplus_{\mathbb{B}}([a_1, b_1] \times \ldots \times [a_n, b_n]), \tau_{\mathbb{B}} e' = \tau_{\mathbb{B}}(e) - \max_i(b_i - a_i)$. This respects trivially our termination criteria as $\max_i(b_i - a_i) > 0$ (except if e is an atom, in which case we would not have split it).

Using this abstract domain in the solving method (Algorithm 1) we retrieve the usual CP solving method on continuous variables. Results detailed in [17] show that this solver terminates and returns a cover over-approximating the solutions.

Relational abstract domains get their names from the fact that they can represent relations between variables. For instance, a linear relation $y \leq x$ can be represented as a polyhedron, but not as an interval. This expressiveness comes at a price, the operators being costlier in relational than in non-relational abstract domains. We adapt here to CP a relational abstract domain, Polyhedra, and present a Reduced Product for CP where Polyhedra and Boxes coexist.

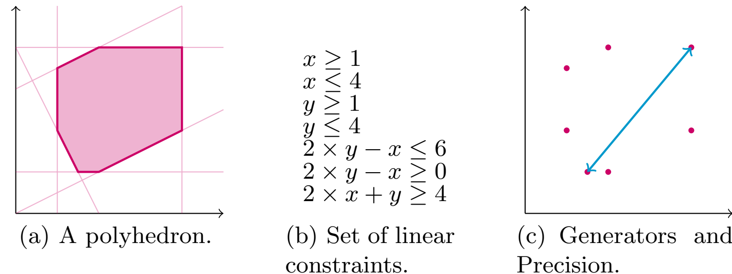
3.2 A Relational Abstract Domain: Polyhedra

The polyhedra domain $\mathbb{P}$ [9] abstracts sets as convex closed polyhedra.

Definition 5 (Polyhedron). *Given a set of linear constraints $\mathcal{P}$, the convex set of $\mathbb{R}^n$ points satisfying all the constraints in $\mathcal{P}$ is called a convex polyhedron.*

Modern implementations [12] generally follow the "double description approach" and maintain two dual representations for each polyhedron: a set of linear constraints and a set of generators. A generator is either a vertex or a ray of the polyhedron. A ray is a vector representing a direction in which the polyhedron is unbounded: starting from any point within the polyhedron, all the points in the direction of the ray remain within the polyhedron. However in practice, polyhedra are bounded in a constraint solver, hence they do not feature rays.

Figure 2 illustrates the different representations for a same polyhedron. The graphical representation 2(a), the set of linear constraints 2(b), and the generators and the maximal distance between two generators 2(c).



(a) A polyhedron. (b) Set of linear constraints. (c) Generators and Precision.

Fig. 2. Different representations for the polyhedra.

The double description is useful because classic polyhedra operators [12] are generally easier to define on one representation rather than the other. This also holds for the operators we introduce here for CP. We define the initialization and the consistency of a polyhedron on the set of linear constraints. The size function is defined on generators and the split operator relies on both representations.

Propagation is an important operator to effectively reduce the search space. In the following, we will consider a weak form of consistency for polyhedra: the non-linear constraints are ignored (not propagated), only the linear constraints are considered.

Definition 6 (Polyhedral consistency). *Let $\mathcal{C}_l$ be a set of linear constraints, $\mathcal{C}_{nl}$ a set of non-linear constraints, and $\mathcal{C} = \mathcal{C}_l \cup \mathcal{C}_{nl}$. The consistent polyhedron for $\mathcal{C}$ is the smallest polyhedron including the solutions of $\mathcal{C}_l$.*

With this weak definition, the consistent polyhedron given a set of constraints always exists and is unique. This simple consistency definition is sufficient when using the polyhedron in the Box-Polyhedra Reduced Product. Note that higher level consistencies could be defined to propagate non-linear constraints, using for instance quasi-linearization [15], linearization of polynomial constraints [13], cutting planes, or computing the hull box, to name a few. Our consistency can be directly computed by adding the constraints to the polyhedron representation.

Proposition 1 (Polyhedral consistency). *The polyhedral consistency returns an over-approximation of the solutions*

Proof. Assume that it does not return an over-approximation, then there exists a polyhedron $P \in \mathbb{P}$, a set of constraints $\mathcal{C}$, the corresponding consistent polyhedron $P_{\mathcal{C}}$, and a solution $s \in P$ such that $s \notin P_{\mathcal{C}}$. Necessarily, one has $c(s)$ for all non-linear constraints c, because non-linear constraints are not considered. Hence there exists a linear constraint c such that $c(s)$ (because s is a solution of the problem) and $\neg c(s)$ because $s \notin P_{\mathcal{C}}$, which gives a contradiction.

The size function is defined as the maximal Euclidean distance between pairs of vertices. Let $P \in \mathbb{P}$,

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

Finally, the splitting operator for polyhedra can be defined in a similar way to that of boxes, *i.e.* by cutting the polyhedron into two parts according to a linear constraint. But we will not use this operator in the following, and omit here its definition.

4 Reduced Product

The Reduced Product, introduced in [7], is a construction to derive a new, more expressive abstract domain, by combining two or more existing ones. An abstract element of the product comprises one abstract element component from each abstract domain, and represents the intersection of the spaces represented by each component. An operation on the reduced product applies the corresponding abstract operation on each component independently, followed by a reduction phase that communicates information between components to improve their precision (for instance, a Polyhedral component can propagate bound informations to a Box component). Hence, deriving a reduced product from existing abstract domains only requires defining one additional operation, the *reduction*, making the reduced product a powerful and attractive abstract domain transformer.

4.1 Constraint-oriented Reduced Products

We present here a generic way of defining Reduced Products for constraint abstract domains. The idea is to combine domains which are not equivalent in the product, and avoid duplicating constraint propagation in each domain. Thus, we introduce a hierarchy between the two components of the product. We make one of the component a specialized domain, dedicated to one type of constraints only, and the second one, a default domain which will apply to the other constraints. We will refer afterwards to these as the default and the specialized domains. This configuration avoids unnecessary computations on the constraints that are not precise or not cheap to represent on some domains (*e.g.,* $x = \cos(y)$ with the domain of polyhedra or $x = y$ with the domain of intervals). Nevertheless, we still keep the modular aspect of the reduced product: we can still add a new domain on top of a reduced product by defining a reduction operator with each existing component, and an attribution operator which specifies for a constraint c if the new specialized domain is able to handle it.

Definition 7. *Let* $(A_d, \sqsubseteq_d), (A_s, \sqsubseteq_s)$ *be two abstract domains. Let* $\mathcal{C}$ *a set of constraints, we define the product* $A_d \times A_s$*, ordered with* $\sqsubseteq$ *where* A_s *is the specialized domain and* A_d *the default one.*

- *The product* $A_d \times A_s$ *is an abstract domain ordered by component-wise comparison. Let* x_d, y_d *be two elements of* A_d *and* x_s, y_s *be two elements of* A_s*, then* $(x_d, x_s) \sqsubseteq (y_d, y_s) \Longleftrightarrow x_d \sqsubseteq_d y_d \wedge x_s \sqsubseteq_s y_s$.
- *A reduction operator is a function* $\theta : A_d \times A_s \rightarrow A_d \times A_s$ *such that* $\theta(x_d, x_s) = (y_d, y_s) \Longrightarrow (y_d, y_s) \sqsubseteq (x_d, x_s)$.

Algorithm 2: Propagation in a reduced product

```
function ρ(e, c)                    // e: abstract element, c: constraint
    (e_s, e_d) ← e
    if κ(c) then
        e' ← (ρ_s(e_s, c), e_d)
    else
        e' ← (e_s, ρ_d(e, c))
    return θ(e')
```

– *Let c be a constraint, an attribution operator κ is a predicate $\kappa : \mathcal{C} \rightarrow \{true, false\}$ such that $\kappa(c) = true$ if the domain A_s is well suited for the constraint c.*

Note that the reduction operator can be seen as a propagator with respect to the constraint: "belong to both elements of the product". Using the reduced product, the propagation loop given in Algorithm 2 slightly differs from the usual one in CP: for each constraint, either the specialized propagator (ρ_s) or the default one (ρ_d) is applied, according to the result of the attribution operator κ. When all of the constraints have been filtered, we then apply the reduction operator (θ) on the resulting abstract element.

Consider A_d, A_s two abstract domains ordered with inclusion, and $A = A_d \times A_s$ the product abstract domain with A_d the default domain and A_s the specialized one. The consistency in A is defined as follow:

Definition 8 (Product-consistency). *Let $\mathcal{C}$ be a set of constraints such that $\mathcal{C} = \mathcal{C}_d \cup \mathcal{C}_s$ with $\mathcal{C}_d = \{C \in \mathcal{C} \,|\, \neg\kappa(C)\}$, the constraints for the default domain A_d, and $\mathcal{C}_s = \{C \in \mathcal{C} \,|\, \kappa(C)\}$, the constraints for the specialized domain A_s. The product-consistent element for $\mathcal{C}$ is the product of the smallest element of A_d including the solutions of $\mathcal{C}_d$ with the smallest element of A_s including the solutions of $\mathcal{C}_s$.*

Proposition 2 (Product-consistency). *The Product-consistency returns an over-approximation of the solutions.*

Proof. Assume that it does not return an over-approximation, then there exists a product $P \in A$, a set of constraints $\mathcal{C}$, the corresponding consistent product P_C, and a solution $s \in P$ of the problem which has been lost, *i.e.* $s \notin P_C$. Then, there exists a constraint c such that $c(s)$ (because s is a solution of the problem) and $\neg C(s)$ because $s \notin P_C$, which gives a contradiction.

4.2 The Box-Polyhedra Reduced Product

The Box-Polyhedra abstract domain $\mathbb{BP}$ is particularly useful when solving problems which involve both linear and non-linear constraints. Here, the Polyhedra domain is used as a specialized domain working only on the linear subset of

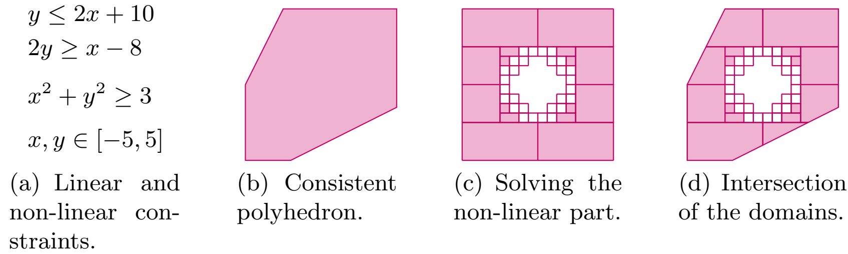


Fig. 3. Example of the Reduced product of Box-Polyhedra.

the problem. We use the Box domain as the default domain to solve the non-linear part of the problem. More precisely, let $\mathcal{C}_l$ bet the set of linear constraints and V_l the set of variables appearing in $\mathcal{C}_l$, and let $\mathcal{C}_{nl}$ be the set of non-linear constraints and V_{nl} the set of variables appearing in $\mathcal{C}_{nl}$. We first build an exact representation of the space defined by $\mathcal{C}_l$ using the Polyhedra domain. By construction, this polyhedron is consistent with respect to $\mathcal{C}_l$ once it is created (conjunctions of linear constraints can be expressed with a convex polyhedron with no loss of precision). In effect, 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 $\mathcal{C}_{nl}$ is solved accordingly.

Figure 3 gives an example of the Box-Polyhedra abstract domain applied on a problem with both linear and non-linear constraints. Figure 3(a) gives the set of constraints, Figure 3(b) the consistent polyhedron (for the linear constraints), Figure 3(c) the union of boxes solving the non-linear constraints, and Figure 3(d) the intersection of both domain 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 $\mathbb{BP}$ are defined only on the box part.

Definition 9 (Box-Polyhedra Consistency). *Let $\mathcal{C} = \mathcal{C}_l \cup \mathcal{C}_{nl}$ with $\mathcal{C}_{nl}$ (resp. $\mathcal{C}_l$) 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 $\mathcal{C}_{nl}$ with the initial polyhedron.*

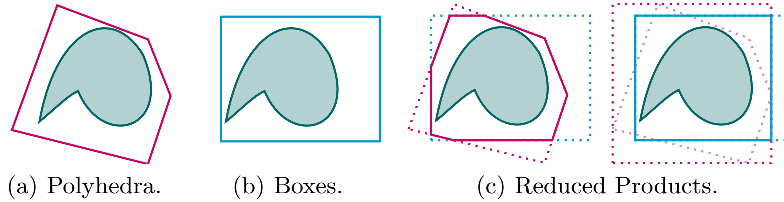
This definition being a particular case of the Product-consistency is thus a correct over-approximation of the solution set.

Let $X = X_b \times X_p \in \mathbb{BP}$ 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):

$$\oplus_{\mathbb{BP}}(X) = \{\oplus_{\mathbb{B}}(X_b) \times X_p\}$$

Finally, the size function is:

$$\tau_{\mathbb{BP}}(X) = \tau_{\mathbb{B}}(X_b)$$



(a) Polyhedra. (b) Boxes. (c) Reduced Products.

Fig. 4. A reduced product for the Box-Polyhedra abstract domains.

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. We do not need any kind of constraint linearization and we reduce the propagation/split phase to one step.

Proposition 3 (Completeness of solving with $\mathbb{BP}$). *The solving method in Algorithm 1 with the $\mathbb{BP}$ abstract domain returns a union of abstract element over-approximating the solution set.*

Proof. The Box-polyhedra consistency computes an over-approximation of the solutions; then, by Definition 10 in [16], the abstract solving method using the $\mathbb{BP}$ abstract domain returns a cover over-approximating the solutions.

5 Experiments

We have implemented the method presented above in the AbSolute constraint solver.[7] It implements the solving method presented in Algorithm 1 and in [16]. This solver is written in OCaml, and uses the APRON numerical abstract domain library [12]. Its current version features a generic implementation of the propagation loop with reduced products, the heuristic for the mixed box-polyhedra abstract domain, and a visualization tool.

Figure 4 shows the results of a Boxes-Polyhedra reduced product. The solution space (in green) is approximated using the polyhedra (resp. boxes), abstract domain on 4(a) (resp. 4(b)). The informations are then shared using the reduced product. The reduced product first transforms the polyhedron into a box by computing its bounding box (this operation is cheap using the generator representation), and then the box into a polyhedron (this step is straightforward as boxes are polyhedra). Finally, the reduction is performed for each abstract element: we propagate constraints from the box into the polyhedron (this step induces no loss of precision) and symmetrically from the polyhedron to the box which gives an over-approximation of their intersection 4(c). In this example, both abstract elements are reduced, but applying the reduced product does not necessarily change both or even either one of the abstract elements.

[7] Available on GitHub `https://github.com/mpelleau/AbSolute`.

Table 1. 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**

In our experiments, we compared our solver with the `defaultsolver` of Ibex 2.3.1 [5], on a computer equipped with an Intel Core i7-6820HQ CPU at 2.70GHz 16GB RAM running the GNU/Linux operating system.

We selected problems from the Coconut benchmark[8]. This benchmark is intended as a test set of continuous global optimization and satisfaction problems and is described in detail in [22]. We have selected problems with only linear constraints, only non-linear constraints or both as this is the main focus of our work according to the constraint language recognized by the AbSolute solver. We have fixed the precision (the maximum size of the solutions, w.r.t. to the size metric of the employed domain) to 10^{-3} for all problems for both solvers.

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 always 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

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

the solution space.[9] Hence, the less elements we have, the faster the computation will be. Furthermore, having fewer elements makes the reuse of the ouput easier.

The first three columns in Table 1 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).

According to the metrics mentioned above, 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 elements whether it satisfies the set of constraints. If this 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 good performances, showing that the time overhead induced by the use of a specialized abstract domain is insignificant when this one is not used for a given problem. 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.

6 Conclusion

In this paper, we introduced a well-defined way of solving constraint problems with several abstract domains. Our idea is to use 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 exactly represented in the specialized domain. This allows us to get the best of both domains, while keeping the solver properties. We have detailed the case of the Polyedra-Boxes product, well suited for problems with linear and non-linear constraints.

[9] Note that AbSolute discriminates the elements in two categories: the ones such that all of the points in them satisfy the constraints, and the one where it is not the case. We have not showed this information in the experiments as Ibex does not do any kind of discrimination on the resulting elements.

The principle is generic enough to add as many specialized domain as one wishes. Integer domains need to be added to the framework, for instance, the Congruence domain, based on constraints of the form: $a \equiv b \pmod{n}$. We also plan to investigate abstract domains efficiently representing global constraints, as, for instance, Octagons and time precedence constraints. These domains could be combined with more basic domains handling any constraint. In general, the Reduced Product construction can be viewed as a way to combine different specific constraint solving mechanisms, within a formal framework to study their properties (soundness, completeness). Ultimately, each CP problem could be automatically solved in the abstract domains which best fits it, as in AI.

Note The research described in this article has been partly funded by the Coverif ANR project 15-CE25-0002-03.

References

1. Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221, 1999.
2. Frédéric Benhamou. Heterogeneous constraint solvings. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, pages 62–76, 1996.
3. 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.
4. Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
5. Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173:1079–1100, 2009.
6. 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.
7. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium of Principles of Programming Languages*, pages 269–282, 1979.
8. 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.
9. 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.
10. David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
11. Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *13th International Symposium on Static Analysis*, pages 18–34, 2006.
12. 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.

13. 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.
14. Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure, December 2004.
15. Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.
16. 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.
17. Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. The octagon abstract domain for continuous constraints. *Constraints*, 19(3):309–337, 2014.
18. 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.
19. Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.
20. Christian Schulte and Guido Tack. Weakly monotonic propagators. In *15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*, pages 723–730, 2009.
21. Joseph Scott. *Other Things Besides Number: Abstraction, Constraint Propagation, and String Variables*. PhD thesis, University of Uppsala, 2016.
22. Oleg Shcherbina, Arnold Neumaier, Djamila Sam-Haroud, Xuan-Ha Vu, and Tuan-Viet Nguyen. Benchmarking global optimization and constraint satisfaction codes. In *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction. Revised Selected Papers*, pages 211–222, 2003.
23. 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.