Combining Constraint Languages via Abstract Interpretation

Pierre Talbot
Université de Nantes, LS2N
44000 Nantes, France
pierre.talbot@univ-nantes.fr

David Cachera
Univ Rennes, Inria, CNRS, IRISA
35000, Rennes
david.cachera@irisa.fr

Éric Monfroy
Université de Nantes, LS2N
44000 Nantes, France
eric.monfroy@univ-nantes.fr

Charlotte Truchet
Université de Nantes, LS2N
44000 Nantes, France
charlotte.truchet@univ-nantes.fr

***Abstract*—Constraint programming initially aims to be a declarative paradigm, but its quest for efficiency is mainly achieved through the development of *ad-hoc* algorithms, which are encapsulated in global constraints. In this paper, we explore the idea of extending constraint programming with abstract domains, a structure from program analysis by abstract interpretation. Abstract domains allow us to efficiently process constraints of the same form, such as linear constraints or difference constraints. This classification by constraint sub-languages instead of sub-problems, makes abstract domains more general and more reusable in many problems. We contribute to the definition of an abstract domain encapsulating a constraint solver in a conservative way w.r.t. constraint programming. We also define a product of abstract domains based on reified constraints and under-approximations. We study a well-known scheduling problem to motivate our approach and experiment its feasibility.**

***Index Terms*—constraint programming, abstract interpretation, cooperation of solvers, mixed domains, reified constraints, under-approximation**

I. Introduction

Constraint programming is a powerful paradigm to model problems in terms of constraints over variables. This declarative paradigm solves many practical problems including scheduling, vehicle routing or biology problems [22], as well as more unusual problems such as in musical composition [29].

In order to be efficient, modern constraint solvers heavily rely on global constraints that are n-ary predicates. They encapsulate efficient solving algorithms for particular sub-problems, for example, $\mathit{alldifferent}(x_1, \ldots, x_n)$ ensures the variables $x_1, \ldots, x_n$ to be all different. Constraint programming provides a solving framework where constraints can be logically composed together, with each one of them effectively dealing with a part of the problem.

A crucial observation is that we can often design a more efficient solving algorithm if it is specialized for a particular problem, and even for a particular class of instances of the problem. Therefore, we shift from a declarative point of view to an algorithmic one, which departs from the initial goal of the constraint paradigm. In particular, this research direction is noticeable in the *global constraint catalog* [4], accounting for more than 400 global constraints.

This work was supported by ANR-15-CE25-0002 Coverif from the French *Agence Nationale de la Recherche*. The *Centre de calcul intensif des Pays de la Loire* (CCIPL) provided the infrastructure to perform the benchmarks.

Although global constraints are important abstractions, constraint solvers usually only implement a dozen of the most important ones. In addition to studying sub-problems, it is interesting to focus on more general *constraint languages*. A constraint language is a conjunction of constraints that all have the same form, for instance:

- `alldifferent` models a language for constraints of the form $x \neq y$ with x and y variables.
- Linear programming is a constraint technology for the language of linear equations of the form $a_1 * x_1 + \ldots + a_n * x_n \geq c$ where a_i and c are constants.
- Difference logic concerns constraints of the form $\pm x \pm y \leq c$.

The advantage of identifying specific constraint languages is to use dedicated solving techniques, such as linear programming.

In this article, we target the combination of such dedicated solvers through *abstract interpretation*. Abstract interpretation is a framework to statically analyse programs by approximating the set of values that can take the variables of a program [7]. We consider a fragment of this theory called *abstract domain*. An abstract domain encapsulates a constraint language such as bound constraints of the form $\pm x \geq c$ (interval domain), linear constraints (polyhedra domain) and difference logic (octagon domain) [18], to name a few. An advantage of abstract domains is their unified formal definitions, as well as *products* to combine several abstract domains. In this paper, we advocate that abstract interpretation provides a formal and compositional framework well suited to combine constraint languages.

Our first contribution is to define a novel abstract domain PP which encapsulates constraint solvers relying on propagator functions—which are the operational form of constraints (Section III-D). Importantly, it implies that the abstract interpretation framework for constraint solving is *conservative* w.r.t. constraint programming, for example global constraints are still supported. A second contribution is a novel product of abstract domains, called the *reified reduced product*, that allows abstract domains to exchange information by means of equivalence constraints (Section V). On theoretical grounds, this reduced product connects *reified constraints* from constraint programming and *under-approximations* from abstract interpretation.

We motivate our work with a well-known scheduling problem (Section II) that can be treated by three abstract domains: PP (Section III-D), integer octagon (Section IV), and the reified reduced product (Section V). These abstract domains fit into the abstract solving framework described in Section III. We experiment on the scheduling problem in Section VI, and show that our approach is feasible, although currently outperformed by modern constraint solvers. We terminate the paper by discussing more precisely the differences between global constraints and abstract domains in Section VII, as well as investigating the connections between our work and *Satisfiability Modulo Theories* (SMT) solvers.

II. Motivating problem: RCPSP

Resource-constrained project scheduling problem (RCPSP) is a problem where we must find a tasks schedule such that resources usage do not exceed some capacities. RCPSP is defined by a tuple $\langle T, P, R\rangle$ where T is the set of tasks, P is the set of precedences among tasks, written $i \ll j$ to indicate that task i must terminate before j starts, and R is the set of resources. Each task $i \in T$ has a duration $d_i \in \mathbb{N}$ and, for each resource $k \in R$, a resource usage $r_{k,i} \in \mathbb{N}$. Each resource $k \in R$ has a capacity $c_k \in \mathbb{N}$ quantifying how much of this resource is available in each instant. The goal is to find a schedule of the tasks T meeting the precedences constraints in P such that, in each instant, the capacities of the resources available are not exceeded. In general, RCPSP is an optimization problem where we search for a solution minimizing the total duration of the schedule.

A constraint model of this problem is to represent each task i with a starting date s_i. All constants and variables are discrete. We define the *temporal constraints* as follows:

$$\forall (i \ll j) \in P,\ s_i + d_i \leq s_j \tag{1}$$

Resources constraints are defined as follows:

$$\forall t \in [0..h-1],\ \forall k \in R,\ (\sum_{i \in T, s_i \leq t < s_i + d_i} r_{k,i}) \leq c_k \tag{2}$$

where h is the horizon of the schedule, which is the latest date a task can end. A simple way to obtain the horizon is to sum the duration of the tasks. In each instant t of the schedule, we constrain the usage of each resource k to not exceed its capacity. A resource is used in an instant if a task using this resource is executed during that instant.

Global constraints emerged as efficient algorithms to solve specialized part of a problem. For instance, resources constraints (2) are solved with a dedicated global constraint called `cumulative`:

$$\texttt{cumulative}_k([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_{k,1}, \ldots, r_{k,n}], c_k)$$

which ensures that tasks do not exceed the capacity c_k of the resource k. We note that `cumulative` only processes one resource $k \in R$ at a time. The proposal in this paper is to use its decomposition into primitive constraints in order to efficiently process classes of constraints in suited abstract domains. To achieve that, we rely on the *tasks decomposition* of `cumulative` as given in [25]:

$$\begin{aligned} &\forall j \in [1..n],\ \forall i \in [1..n] \setminus \{j\}, \\ &\qquad b_{i,j} \Leftrightarrow (s_i \leq s_j \land s_j < s_i + d_i) \end{aligned} \tag{3}$$

$$\forall j \in [1..n],\ r_{k,j} + (\sum_{i \in [1..n] \setminus \{j\}} r_{k,i} * b_{i,j}) \leq c_k \tag{4}$$

Part (3) of the decomposition introduces $n^2 - n$ Boolean variables, where a variable $b_{i,j}$ is true iff the tasks i and j overlap. This is realized by reifying the overlap constraint $s_i \leq s_j \land s_j < s_i + d_i$ into the Boolean variable $b_{i,j}$. For the constraints in (4), we rely on the observation that a task can not be preempted, hence the usage of resources only changes when a task starts. Therefore, for all starting dates s_j, the sum of its consumption of resources $r_{k,j}$ and the resources of the tasks i overlapping with j must not exceed the capacity c_k.

In the following, we consider the generalized version *RCPSP/max* where precedences between two tasks are generalized. The set P contains temporal constraints of the form:

$$\pm s_i \pm s_j \leq c \tag{5}$$

where i, j are tasks and $c \in \mathbb{Z}$ an integer constant. For instance, the constraint $s_2 - s_1 \leq 3$ means that task 2 must start at the latest 3 instants after task 1. Additional constraints such as “at the latest”, “at the earliest”, “exactly after n instants”, “before”, “after” are all instances of this temporal constraint.

All in all, the RCPSP problem is defined over three classes of constraints:

(a) Resources constraints modelled in (4).
(b) Generalized temporal constraints occurring in (3) and (5).
(c) Equivalence constraints bridging (a) and (b) in (3).

In the following, we introduce three abstract domains for these classes: an abstract domain for constraint satisfaction problems (CSP) to treat constraints (a) in Section III, octagon abstract domain for (b) in Section IV, and the reduced product for equivalence constraints in Section V.

III. Abstract interpretation for constraint programming

Abstract interpretation is a framework to statically analyse programs by over-approximating the set of values that can take the variables of the program [7]. We will focus on a fragment of this theory which consists in *abstract domains*.

A. Concrete Domain

A constraint satisfaction problem (CSP) is a tuple $\langle X, D, C\rangle$ where X is a set of variables, $D_i \in D$ the set of values taken by each variable $x_i \in X$, and C a set of relations over variables, called *constraints*. A constraint $c \in C$, defined on the variables $x_1, \ldots, x_n$ is satisfied when $c(v_1, \ldots, v_n)$ holds for all $v_i \in d_i$. The *concrete domain* is a lattice $D^b = \langle \mathcal{P}(D), \subseteq\rangle$

ordered by inclusion, and each CSP $\langle X, D, C\rangle$ has an element in D^b representing its set of solutions:

$$\{D' \mid D' \subseteq D \text{ and all } c \in C \text{ satisfied}\} \in D^b$$

We write $sol^b(c)$ the set of solutions of a single constraint. This extensional representation of a CSP is not necessarily computable (for example on real numbers). We abstract the concrete domain to an *abstract domain* which is computable, but which can over-approximates (contains elements that are not solutions) or under-approximates (contains only solutions but not all) the set D^b.

B. Abstract Domain

In abstract interpretation, an abstract domain is a partially ordered set equipped with useful operations for programs analysis. This notion has been adapted to constraint programming, where some operators are reused (*e.g.*, join and transfer function) and some are new (*e.g.*, state and split) for its application to constraint solving [20]. In this paper, "abstract domain" will refer to this modified notion of abstract domain for constraint programming that is defined as follows. We use the set $K = \{true, false, unknown\}$ to represent elements of Kleene logic (for instance, $false \wedge unknown = false$ and $true \wedge unknown = unknown$).

Definition 1 (Abstract domain)**.** *An abstract domain for constraint programming is a lattice $\langle Abs, \leq\rangle$ where Abs is a set of computer-representable elements equipped with the following operations:*

- *$\bot$ is the smallest element and, if it exists, $\top$ the largest.*
- *$\sqcup : Abs \times Abs \to Abs$ is the join operation between two elements.*
- *$\gamma : Abs \to D^b$ is a monotonic concretization function mapping an abstract element to its set of solutions.*
- *$state : Abs \to K$ gives the state of an element: true if the element satisfies all the constraints of the abstract domain, false if at least one constraint is not satisfied, and unknown if satisfiability cannot be established yet.*
- *$\vDash$: $Abs \times C$ is a deduction relation, called the entailment, where $a \vDash c$ holds iff we can deduce the constraint c from a, i.e. if $\gamma(a) \subseteq sol^b(c)$.*
- *$[\![.]\!] : C \to Abs$ is a partial function transferring a logical constraint to an element of the abstract domain. This function is not necessarily defined for all constraints since an abstract domain efficiently handles a delimited constraint language.*
- *$closure : Abs \to Abs$ is an extensive function ($\forall x, x \leq closure(x)$) which eliminates inconsistent values from the abstract domain. In constraint programming, closure is known as propagation.*
- *$split : Abs \to \mathcal{P}(Abs)$ divides an element of an abstract domain into a finite set of sub-elements.*

We refer to the ordering of the lattice L as $\leq_L$ and similarly for any operation defined on L, unless no confusion is possible. Finally, we note that constraints are encapsulated inside the abstract domain via the $[\![.]\!]$ function, thus the $closure$ operator can be seen as an algorithm propagating the constraints belonging to this abstract domain. We now illustrate this definition on the interval abstract domain.

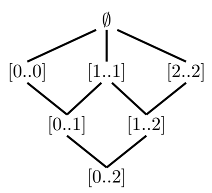
C. Interval Abstract Domain

We denote by $\mathbb{A}$ either the set of integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$ or floating point numbers $\mathbb{F}$. We add the positive and negative infinite elements ∞ and $-\infty$ into $\mathbb{A}$. We also define the successor function as

$$\begin{array}{ll} succ(a) = a + 1 & \text{if } a \in \mathbb{Z} \setminus \{-\infty, \infty\} \\ succ(a) = a & \text{otherwise} \end{array}$$

The successor of infinity, rational and floating point numbers either does not exist or is not computable, in which cases it is defined by the identity function.

An interval is a pair $(l, u) \in \mathbb{A}^2$ of the lower and upper bounds, written $[l..u]$, defined as $\gamma([l..u]) = \{x \mid l \leq x \leq u\}$ where $x \in \mathbb{R}$ on rational and floating point numbers, and $x \in \mathbb{Z}$ on integers. The lattice of intervals is $I = \langle\{[l..u] \mid \forall l, u \in \mathbb{A}\}, \leq, \bot, \top, \sqcup\rangle$, ordered by set inclusion $\leq \triangleq \subseteq$, with bottom $\bot \triangleq [-\infty, \infty]$ the set of all elements, top $\top \triangleq \{\}$, and join $\sqcup \triangleq \cap$ defined by set intersection. An example of this lattice for the set $\{0, 1, 2\}$ is depicted as:



Let V be a set of variables. The interval abstract domain $\mathcal{I}$ is the powerset of the indexed set $X, Y \in \mathcal{P}(V \times I)$ s.t.:

- The order is the Smyth order, where an element Y is greater than X if the interval of all the variables in Y is included in X:
$$X \leq Y \text{ if } \forall (x, i) \in X,\ \exists (y, j) \in Y,\ x = y \implies i \leq_I j$$
- $\bot \triangleq \{\}$ and $\gamma(X) = \{(x, \gamma(i)) \mid (x, i) \in X\}$.
- Let $Z = \{(x, i \sqcup_I j) \mid (x, i) \in X,\ (x, j) \in Y\}$ be the intersection of X and Y, X' the set of the elements with a variable appearing in X and not in Y, and conversely for Y'. Then $X \sqcup Y \triangleq Z \cup X' \cup Y'$.

Then $\langle \mathcal{I}, \leq, \bot, \sqcup\rangle$ is a lattice, which is a standard result (see [11]). We make it into an abstract domain by defining the corresponding operations:

- We have
$$state(X) = \begin{cases} false & \text{if } \exists (x, i) \in X,\ i = \top_I \\ true & \text{otherwise} \end{cases}$$
- Let $x \in V$, $b \in \mathbb{A}$, then $[\![x \leq b]\!] = \{(x, [-\infty..b])\}$ and $[\![x \geq b]\!] = \{(x, [b..\infty])\}$.
- $X \vDash c \triangleq X \geq [\![c]\!]$
- $closure(X) = X$
- A possible split is by bissection on the domain of a variable $(x, [l..u]) \in X$ s.t. $l < u$. Let $m = \lfloor (l + u)/2 \rfloor$, then we have:
$$split(X) = \{X \sqcup \{(x, [l, m])\}, X \sqcup \{(x, [succ(m), u])\}\}$$

Notice the use of $succ$ to define $split$ generically over continuous and discrete domains.

Example 2. *We illustrate these operations with three elements of the interval abstract domain:* $X = \{(x, [0..1]), (y, [0..1])\}$, $Y = \{(x, [0..2]), (y, [0..2])\}$ *and* $Z = \{(x, [3..4]), (y, [3..4])\}$. *For instance, we have* $X \sqcup Y = X$, $state(X \sqcup Z) = false$, $X \geq Y$, *and* X *and* Z *are not ordered.*

Entailment. In contrast to the transfer function, we forbid the entailment to over-approximate a constraint, thus we cannot define it as $X \vDash c \triangleq X \geq [\![c]\!]$ in the general case. Consider for example $\{(x, [5.0..6.0])\} \vDash x > 5.0$ on floating point intervals, and the over-approximation $[\![x > b]\!] = [\![x \geq b]\!]$. The constraint $x > 5.0$ cannot be represented exactly, and is over-approximated to $x \geq 5.0$. Although the entailment holds on this over-approximation, it has the solution $\{(x, [5.0..5.0])\}$ which is not included in $sol^b(x > 5.0)$. However, notice that it is correct to under-approximate $x > 5.0$, for example to $x \geq 5.1$, in which case if the entailment holds with the under-approximation, it also holds with the initial constraint. These considerations are important in Section V, when we use the entailment as a guard to add more constraints into the problem. Similar concerns were tackled for finite domains in CC(FD) [5].

Of course, this domain alone is not very interesting since it only takes care of bound constraints. We now equip intervals with a set of constraint functions in order to abstract a CSP.

D. PP Abstract Domain

We turn a logical constraint $c \in C$ into an extensive function $p : \mathcal{I} \to \mathcal{I}$, called *propagator*, over the interval abstract domain. For example, given $d = \{(x, [1..2]), (y, [2..3])\} \in \mathcal{I}$ and the constraint $x \geq y$, a propagator $p_\geq$ associated to $\geq$ gives $p_\geq(d) = \{(x, [2..2]), (y, [2..3])\}$. We notice that this propagation step is extensive, e.g., $d \leq p_\geq(d)$. Beyond extensiveness, a propagator must also be sound, *i.e.* it should not remove solutions of the logical constraint, in order to guarantee the correctness of the solving algorithm.

We associate to each propagator p a $state_p$ function which is defined similarly to the one of abstract domain. In particular, an element d is a solution of p if $state_p(d) = true$.

We now define the lattice $Pr = \langle \mathcal{P}(Prop), \subseteq \rangle$ where $Prop$ is the set of all propagators (extensive and sound functions). The lattice of all propagation problems (PP)—a CSP with propagators instead of logical constraints—is given by the Cartesian product $PP = \mathcal{I} \times Pr$. An element $\langle d, P \rangle \in PP$ is greater than $\langle d', P' \rangle \in PP$ if its variables' domains are smaller or it has more propagators. The join is defined as $\langle d, P \rangle \sqcup \langle d', P' \rangle = \langle d \sqcup_I d', P \cup P' \rangle$.

We define the necessary operations to turn $\langle d, P \rangle \in PP$ into an abstract domain. First we have $state$:

$$state(\langle d, P \rangle) = state(d) \wedge \begin{cases} true & \text{if } \forall p \in P,\ state_p(d) = true \\ false & \text{if } \exists p \in P,\ state_p(d) = false \\ unknown & \text{otherwise} \end{cases}$$

which intuitively means that we reached a solution if d is a solution for all propagators in P.

The concretization is defined as:

$$\gamma(\langle d, P \rangle) = \{\gamma(d') \mid d' \geq d \wedge state(\langle d', P \rangle) = true\}$$

The function $[\![c]\!]$ associates the constraint c to its propagator p_c and state function $state_c$. We note that global constraints are still supported within this model.

The entailment is then defined as follows:

$$\langle d, P \rangle \vDash c \text{ if } state_c(d) = true$$

If the constraint c is satisfied in d, then we can conclude that c does not remove solutions from $\langle d, P \rangle$. We note that we do not need the propagator of the constraint c but just its state function. This is particularly useful to ask if the negation of a global constraint is entailed (since propagators of the negation of global constraints are not usually trivial, see [3]).

Given $\langle d, \{p_1, \ldots, p_n\} \rangle \in PP$, the *propagation step* is realized by computing the fixpoint of $p_1 ... p_n$ altogether. We note $closure : PP \to PP$ the function computing this fixpoint. There are many possible implementation of $closure$ as shown in [1], [23], [28]. We leave this choice to the implementer of this abstract domain.

The split function can be defined similarly to that of the abstract domain of intervals.

E. Solving Algorithm

We now present a generic abstract constraint solving algorithm, and some conditions for termination.

```
1: function solve($a \in Abs$)
2:     $a \leftarrow$ closure(a)
3:     if state(a) = true then return $\{a\}$
4:     else if state(a) = false then return $\{\}$
5:     else
6:         $\langle a_1, \ldots, a_n \rangle \leftarrow$ split(a)
7:         return $\bigcup_{i=0}^{n}$ solve($a_i$)
8:     end if
9: end function
```

This algorithm follows the usual solving pattern in constraint programming which is *propagate and search.* We infer as much information as possible with $closure$, and then we divide the problem into sub-problems with $split$. We rely on $state$ for the base cases defined when we reach a solution or an inconsistent node. Note that the abstract domain $a \in Abs$ can be a composition of several abstract domains through reduced product (see Section V).

Various termination conditions can be designed for $solve$. We present one that is usually fulfilled in constraint solver.

Property 3. *Let* $a \in Abs$, *then* $solve(a)$ *terminates if, given a chain* $a_1 < \ldots < a_n < \ldots$ *with* $a_i \in Abs$ *produced by recursive calls on solve,* $state(a_n)$ *is equal to* $true$ *or* $false$.

As an example, we give sufficient conditions on the abstract domain PP to match Property 3:

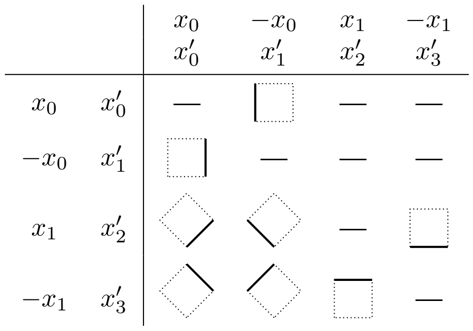


Fig. 1: Correspondence DBM/octagon in dimension 2.

Proposition 4. *$solve(\langle d, P\rangle \in PP)$ terminates if:*

(a) *The number of variables does not increase.*
(b) *Every interval is bounded, i.e., $\forall(x, [l..u]) \in d,\ l, u \notin \{-\infty, \infty\}$.*
(c) *For each propagator $p \in P$, we have $state_p(d) \neq unknown$ whenever all variables in d are*
  - *assigned to a singleton interval (case $\mathbb{Z}$).*
  - *smaller than a precision $\tau \in \mathbb{A}$ (case $\mathbb{Q}$ or $\mathbb{F}$).*

Proof. Condition (a) is met in the usual case as $closure$ and $split$ keeps the number of variables unchanged. We can check that condition (b) is met by verifying that the initial element $\langle d, P\rangle$ only contains bounded domains. We note that it depends on the model of the user. Condition (c) must be checked individually for each propagator (see, e.g., [14], [28]).

If $split$ is strictly extensive, then each sequence of recursive $solve$ calls produces a strictly increasing sequence of elements. Hence, the algorithm eventually terminates since $state$ is defined on top of $state_p$ for each propagator p, which eventually terminates by condition (c). □

Other termination conditions include bounding the depth, the number of nodes, solving time, but these do not usually preserve completeness (notice that completeness may not be preserved on continuous domains anyway).

The function $solve$ on the PP abstract domain is not different from the usual solving algorithms in constraint solvers. The advantage is to be totally generic w.r.t. an abstract domain, which can be defined over continuous or discrete domains, but also works with other kind of constraint solvers such as in linear programming (Polyhedra domain) or difference logic (Octagon domain).

IV. Integer Octagon

A. Definition

The integer octagon abstract domain is defined on a set of variables $(x_0, \dots, x_{n-1})$ and a conjunction of *octagonal constraints* of the form:

$$\pm x_i \pm x_j \leq d$$

where $d \in \mathbb{Z}$ is a constant. By extending the set of variables to $(x'_0, \dots, x'_{2n-1})$, Miné [17] gives a transformation of these constraints into *potential constraints* of the form $x'_i - x'_j \leq d$ where only positive variables occur. A set of potential constraints can be solved in cubic time by the shortest paths Floyd-Warshall algorithm. We represent these constraints in a difference bounded matrix (DBM) corresponding to the octagon. A DBM is a matrix m where an element $m_{i,j}$ is the constant d of the potential constraint $x'_j - x'_i \leq d$. We note that whenever $j/2 > i/2$ (where / is the integer division), the element $m_{\bar{j},\bar{i}}$ is redundant with $m_{i,j}$. In the literature, a matrix with equal redundant elements is said *coherent*. This property is made implicit in the implementations, where the redundant elements are not represented at all. We however keep the full matrix in the definitions for sake of clarity.

A geometrical intuition of a n-dimensional octagon is to understand it as an intersection of n-dimensional boxes. We give an example in 2 dimensions on Figure 1. The constraints correspond to each side of the octagon, the non-rotated box models the bound constraints ($x \leq v \wedge x \geq v$), and the box rotated at 45° models the potential constraints. The lines and columns of the matrix are annotated with their corresponding variables, and we depict each element of the matrix with the side of the octagon it represents. Given the DBM index (i, j), a useful operation is to retrieve the index of the opposite side of the octagon with $(\bar{\imath}, \bar{\jmath})$ where

$$\bar{\imath} = \begin{cases} i+1 \text{ if } i \text{ is even} \\ i-1 \text{ if } i \text{ is odd} \end{cases}$$

and similarly for $\bar{\jmath}$.

We transform a set of octagonal constraints to a set of potential constraints with the following rewriting function $\rightsquigarrow$ (with $i \neq j$):

$$\begin{array}{lcll}
x_i \geq d & \rightsquigarrow & x'_{2i+1} - x'_{2i} & \leq -2d \\
x_i \leq d & \rightsquigarrow & x'_{2i} - x'_{2i+1} & \leq 2d \\
x_i - x_j \leq d & \rightsquigarrow & x'_{2i} - x'_{2j} & \leq d \\
x_i + x_j \leq d & \rightsquigarrow & x'_{2i} - x'_{2j+1} & \leq d \\
-x_i - x_j \leq d & \rightsquigarrow & x'_{2i+1} - x'_{2j} & \leq d \\
-x_i + x_j \leq d & \rightsquigarrow & x'_{2i+1} - x'_{2j+1} & \leq d
\end{array}$$

Example 5. *To illustrate the relations between octagonal constraints, the DBM and its geometric representation, we consider the following constraints:*

$$\begin{array}{ll}
x_0 \geq 1 \wedge x_0 \leq 3 & x_1 \geq 1 \wedge x_1 \leq 4 \\
x_0 - x_1 \leq 1 & -x_0 + x_1 \leq 1
\end{array}$$

Bound constraints on x_0 and x_1 are represented by the yellow box on Figure 2b, and octagonal constraints by the green box (resp. light and dark gray). The intersection of these two boxes is depicted on Figure 2c.

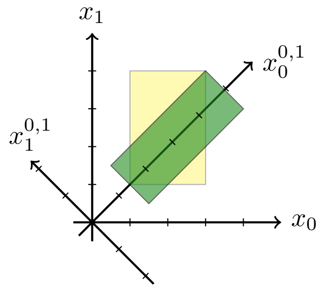
The potential constraint associated to $-x_0 + x_1 \leq 1$ is $x'_1 - x'_3 \leq 1$, and is represented in the DBM entry $dbm_{3,1} = 1$ (Figure 2a). Using Figure 1, we can find that the DBM entry $(3, 1)$ represents the upper left side of the octagon.

B. Operations

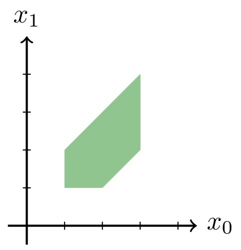
We rely on the matrix representation of an octagon to define its operations. Let m and m' be DBMs of dimension n and N the set $\{0, \dots, 2n-1\}$. A matrix is a set of indexed elements $\{d^{i,j} \mid i, j \in N\}$.

	x'_0	x'_1	x'_2	x'_3
x'_0	0	-2		
x'_1	6	0		
x'_2	1	-2	0	-2
x'_3	7	1	8	0

(a) DBM



(b) Octagon as intersection of two boxes



(c) Octagon

Fig. 2: DBM and its representation as the intersection of two boxes.

The closure operator is obtained with the Floyd-Warshall algorithm. This algorithm is extensively studied in the literature so we do not recall it here (see [17], [27]). In particular, we consider the incremental variant [2], [6] that allows us to update the DBM with an octagonal constraint with a time complexity of $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$ for the general closure. Given a DBM m, we write its closed DBM $m^* = closure(m)$.

The smallest octagon $\perp$ is defined by the matrix $\{\infty^{i,j} \mid i,j \in N\}$.

The concretization of a DBM is the set of points included in the octagon:

$$\gamma(m^*) = \{(v_1, \ldots, v_n) \in \mathbb{Z}^n \,|\, \forall i,j \in N,\ v_j - v_i \leq m_{i,j}\}$$

Next, we define the operation $\sqcup$ defined as follows:

$$m \sqcup m' = \{min(m_{i,j}, m'_{i,j})^{i,j} \mid i,j \in N\} \tag{6}$$

This join operation is the intersection of two octagons obtained by taking the minimal coefficients of these two DBMs. Intuitively, if a coefficient is smaller, it means that the interval between two parallel sides of the octagon is narrower. The order $m' \leq m$ is equivalent to $m' \sqcup m = m$.

Next, we have the operation $state$:

$$state(m) = \begin{cases} false & \text{if } \exists i,j \in N,\ m_{i,j} + m_{\bar{\imath},\bar{\jmath}} < 0 \\ true & \text{if } \forall i,j \in N,\ m_{i,j} + m_{\bar{\imath},\bar{\jmath}} \geq 0 \\ & \text{and } m \text{ is closed} \\ unknown & \text{otherwise} \end{cases} \tag{7}$$

The octagon is inconsistent ($false$) if two sides in parallel are inverted, which translates in a negative sum of their coefficients[1]. It is $true$ if the DBM is closed and it is not inconsistent. In all other cases, the state of the octagon is unknown.

The transfer operation in octagon is only defined for octagonal constraints. Let a, b be the indices of the octagonal variables and i, j of the potential variables, we have the following rewriting:

$$\pm x_a \pm x_b \leq d \rightsquigarrow x'_i - x'_j \leq d'$$

[1]In the DBM representation, the sign of the coefficient of a side—representing a lower bound—must be negated to obtain its Euclidian coordinate.

and the following transfer operation:

$$[\![\pm x_a \pm x_b \leq d]\!] = \perp \sqcup \{d'^{i,j}\}$$

We rely on the rewriting from octagonal to potential constraints to set the coefficient d' in the DBM.

The entailment is defined as $m \vDash c \triangleq state(m \sqcup [\![c]\!]) = true$ since the transfer function does not over-approximate constraints (it is exact). The entailment is efficiently performed in constant time because we only have one octagonal constraint to check. In the continuous case, we might support over-approximation of constraints of the form $\pm x \pm y < d$, relaxed to $\pm x \pm y \leq d$, which is not correct in the context of the entailment. The solution is to under-approximate such constraints with $\pm x \pm y \leq d - \delta$ where $\delta \in \mathbb{A}$ is small enough.

Finally, we define a split operator based on the geometric intuition of octagon, which consists in dividing into sub-problems that minimize the distance between two sides of the octagon. Let a, b the index of the DBM maximizing the expression $|m_{i,j} + m_{\bar{\imath},\bar{\jmath}}|$ for every $i,j \in N$, and $m_{i,j} \geq m_{\bar{\imath},\bar{\jmath}}$. Let $mid = (m_{a,b} + m_{\bar{a},\bar{b}})/2$, then by bisection on the domain, we have:

$$split(m) = \{m \sqcup \{mid^{a,b}\}, m \sqcup \{(mid+1)^{\bar{a},\bar{b}}\}\} \tag{8}$$

In our case, the entry $m_{a,b}$ represents the upper bound of a side of the octagon and $m_{\bar{a},\bar{b}}$ its lower bound.

V. Reduced product with reified constraints

We now have two abstract domains useful for the RCPSP problem: propagation problem PP and octagon Oct. The remaining step is to connect these two domains so they can communicate, which is achieved by reified constraints.

A. Reified Constraint

A reified constraint has the form $c_1 \Leftrightarrow c_2$ where the state of c_1 is required to be equivalent to the one of c_2. The main strength of this equivalence is that c_1 and c_2 are not necessarily represented in the same abstract domain. Therefore, this mechanism is suited to exchange information between two abstract domains.

We have reified constraints (in eq. (3) of the RCPSP model) where Boolean variables can be represented in the PP abstract domain, and constraints of the form $s_i \leq s_j \wedge s_j < s_i + d_i$ in the octagon abstract domain.

Using the entailment operator, we define a generic propagator for equivalence constraints between two abstract domains. Without loss of generality, we rely on PP and Oct to illustrate the definition. Let $c_1 \Leftrightarrow c_2$ an equivalence constraint where $[\![c_1]\!]_{PP}$ and $[\![c_2]\!]_{Oct}$ are defined, then we have:

$$prop_{\Leftrightarrow}(b, o, c_1 \Leftrightarrow c_2) \triangleq \begin{cases} b \vDash_{PP} c_1 \implies (b, o \sqcup [\![c_2]\!]_{Oct}) \\ b \vDash_{PP} \neg c_1 \implies (b, o \sqcup [\![\neg c_2]\!]_{Oct}) \\ o \vDash_{Oct} c_2 \implies (b \sqcup [\![c_1]\!]_{PP}, o) \\ o \vDash_{Oct} \neg c_2 \implies (b \sqcup [\![\neg c_1]\!]_{PP}, o) \\ (b, o) \text{ otherwise} \end{cases}$$

Proposition 6. *$prop_{\Leftrightarrow}(b, o, c_1 \Leftrightarrow c_2)$ is a sound propagator (it does not remove solutions).*

Proof. It follows from the fact that $\vDash$ is required to under-approximate c_1 and c_2, therefore if one is entailed, adding it (or an equivalent constraint) into the abstract domain does not remove solutions. □

This propagator can be generalized to any equivalence constraint of the form $c_1 \wedge \ldots \wedge c_n \Leftrightarrow c'_1 \wedge \ldots \wedge c'_m$ by extending the entailment operator on conjunctions.

B. *Products of PP and Oct*

The *direct product* is an abstract domain combining two abstract domains. It is a Cartesian product of two abstract domains with operators redefined on this product. We can define it on $PP \times Oct$ as follows:

$$\bot = (\bot_{PP}, \bot_{Oct})$$

$$(b, o) \sqcup (b', o') = (b \sqcup_{PP} b', o \sqcup_{Oct} o')$$

$$(b, o) \vDash c = b \vDash_{PP} c \text{ or } o \vDash_{Oct} c$$

$$state((b, o)) = state_{PP}(b) \wedge state_{Oct}(o)$$

$$[\![c]\!] = \begin{cases} ([\![c]\!]_{PP}, [\![c]\!]_{Oct}) \\ ([\![c]\!]_{PP}, \bot_{Oct}) & \text{if } [\![c]\!]_{Oct} \text{ is not defined} \\ (\bot_{PP}, [\![c]\!]_{Oct}) & \text{if } [\![c]\!]_{PP} \text{ is not defined} \end{cases}$$

$$closure((b, o)) = (closure(b), closure(o))$$

$$split((b, o)) = \{(b', o') \mid b' \in split_{PP}(b), o' \in split_{Oct}(o)\}$$

The problem of the direct product is that abstract domains do not exchange information, which is especially important in $closure$. The *reduced product* augments direct product with information exchange. There exists many different reduced products according to how the information are exchanged. We refer to [18] for additional explanations on both products.

We introduce a novel reduced product for abstract domains disjoint on their variable sets and communicating exclusively via reified constraints. We add to the direct product a set R of reified constraints, and redefine the closure operator on the reduced product $PP \times Oct$:

$$closure_R(b, o, R) = (\textstyle\bigsqcup_{r \in R} prop_{\Leftrightarrow}(b, o, r), R)$$

$$closure((b, o, R)) = (closure_R(closure(b), closure(o), R))$$

solver	feas. (%)	opt. (%)	unsat. (%)	Δ_{LB}
sm_j10	**69.26**	**69.26**	**30.74**	**0.00**
AbSolute	69.26	65.93	30.37	0.56
GeCode	69.26	**69.26**	**30.74**	0.00
Chuffed	69.26	**69.26**	**30.74**	0.00
sm_j20	**68.15**	**68.15**	**31.85**	**0.00**
AbSolute	68.15	29.63	28.15	5.01
GeCode	68.15	53.33	28.15	1.83
Chuffed	68.15	**68.15**	**31.85**	0.00
sm_j30	**68.52**	**68.52**	**31.48**	**0.00**
AbSolute	66.30	22.96	29.63	5.58
GeCode	67.78	42.96	28.52	2.84
Chuffed	68.52	**65.56**	**31.48**	0.46
ubo100	**86.67**	**86.67**	**13.33**	**0.00**
AbSolute	58.89	15.56	11.11	6.96
GeCode	66.67	22.22	**12.22**	6.78
Chuffed	86.67	**68.89**	10.00	0.31
ubo200	**88.89**	**88.89**	**11.11**	**0.00**
AbSolute	52.22	8.89	**8.89**	6.58
GeCode	54.44	28.89	**8.89**	9.37
Chuffed	80.00	**62.22**	4.44	7.17

TABLE I: Experiments on 5 sets of instances.

Let e be an element of the reduced product, the closure operator can be applied until we reach the fixpoint $closure(e) = e$. The function $closure_R$ performs the exchange of information by applying the propagators $prop_{\Leftrightarrow}$ on both abstract domains.

VI. Implementation and experiments

We have implemented the three abstract domains described in this paper in the OCaml constraint solver `AbSolute`. Our code and benchmarks are available on `github.com/ptal/AbSolute/tree/ictai2019`. This solver is a prototype that does not aim to compete with existing solvers, but to test new ideas borrowed from abstract interpretation. In this section, we evaluate our approach on a set of classical RCPSP/max benchmarks.

We consider 5 sets of instances from the PSPLIB library, a suite of benchmarks for RCPSP and RCPSP/max [16]:

- `sm_j10`, `sm_j20` and `sm_j30` contain each 270 instances, and respectively 10, 20 and 30 activities and 5 resources [15].
- `ubo100` and `ubo200` contain each 90 instances of 100 and 200 activities with 5 resources [13].

We fix the maximal solving time to 10 minutes for all instances. The experiments are performed on an Intel(R) Xeon(TM) E5-2630 V4 running at 2.20GHz on GNU Linux. The solver `AbSolute` is compiled with the version `4.07.1+flambda` of the OCaml compiler.

We compare `AbSolute` to `GeCode` [24], a state of the art constraint solver, and `Chuffed` [19] the state of the art constraint solver for scheduling problems including RCPSP/max. For all three solvers, we use a classical *branch and bound* algorithm without restarts. The *search strategy* is based on the starting dates of the tasks: we select the variable with the smallest lower bound, and assign this variable to this bound. The specificities of each solver are as follows:

- `AbSolute` relies on the reduced product defined in V.

	Global constraints	**Abstract domains**
Constraint language	Heterogeneous	Homogeneous
Interactivity	Static	Dynamic
Composition of constraints	In a logical formula	By join $\sqcup$ if homogeneous, by reduced product otherwise
Entailment	Usually not supported	Operator $\vDash$

TABLE II: Comparison between global constraints and abstract domains.

- `GeCode 6.0.1` treats the resource constraints with the global constraint `cumulative` implemented with a *timetabling* algorithm [30].
- `chuffed 0.10.0` is run on the same model than `AbSolute`. This solver implements a hybrid solving technique between SAT solver and constraint solver, which is very successful on scheduling problems.

We give preliminary results on Table I where the columns *feas.* is the percentage of feasible instances (at least one solution is found), *opt.* the proven optimal instances and *unsat.* the proven unsatisfiable instances. For feasible instances, the column Δ_{LB} gives the difference between the solutions found and the best known lower bounds. The first line of each set of instances (starting with `sm_j` and `ubo`) contains the proportion of feasible, optimal and unsatisfiable instances.

`AbSolute` and `GeCode` are largely outperformed by `Chuffed` which relies on the successful conflicts analysis from SAT solving. Nevertheless, we notice that `AbSolute` is efficient to prove unsatisfiability; it is better than `GeCode` on `sm_j30` and than `Chuffed` on `ubo100` and `ubo200`. The reason is that temporal reasoning with octagon implements path consistency which efficiently detects early inconsistency. Overall, `AbSolute` is almost as good as `GeCode` at finding feasible solutions, but stays behind to prove optimality. One reason is the number of nodes processed by second in `AbSolute` which is about 10 times slower than `GeCode`, thus exploring a much smaller portion of the search tree. In this respect, many optimisations remain to be done in the implementation of `AbSolute`.

VII. Related work

Solving algorithms for difference constraints are not new, they appear in various works such as temporal constraint networks [9], octagon in abstract interpretation [17], and difference logic in *Satisfiability Modulo Theories* (SMT) solvers [21].

Feydy *et al.* [12] encapsulate difference constraints and reified constraints into a global constraint. Their approach relies on the Bellman-Ford algorithm for satisfiability, which has a complexity of $\mathcal{O}(n \log n + m)$ where n is the number of variables and m the number of difference constraints. For sparse graphs, it allows them to add a single constraint in $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$ for octagons. However, checking the entailment of p constraints has a cost of $\mathcal{O}(n \log n + m + p)$, in comparison to $\mathcal{O}(p)$ with octagons. Moreover, the split strategy (eq. 8) cannot be efficiently implemented since it relies on computing over all shortest paths. Besides these complexity results, reified constraints are encapsulated inside the global constraint whereas it is an orthogonal concern in abstract domains, treated with another construct (the reified reduced product). We also note that such encapsulation of reified constraints inside global constraints is not usually defined. It is also limited by the static framework of global constraints, where one must know in advance which constraints might be added during solving; this is another issue for programming the split and more interactive applications. We summarize these differences on Table II. Note that the abstract domain PP supports heterogeneous constraints, but this is not usually the case in abstract interpretation.

SMT is a research field with extensive literature on combining logical theories (see e.g. [26]). A central concern in SMT is the study of properties of theories (such as stable-infiniteness and convexity) and what are the properties conserved when combining two theories. In practice, the combination of theories in solvers is often *ad-hoc* and exchange of information is complex in order to achieve high efficiency. In our framework, the implementation is almost direct from the definition of abstract domains and reduced product, although currently more specialized (disjoint set of variables) and less efficient than SMT solvers. Several recent works [8], [10] attempt to relate abstract interpretation and SMT solvers, in particular [8] which views Nelson-Oppen theory combination procedure in terms of a specific reduced product. More work is needed in order to draw clear links between SMT, abstract interpretation and constraint programming—this paper focusing on the last two frameworks. In particular, we observe that the reified reduced product can be understood as $SAT(PP \times Oct)$ with $\times$ the direct product and SAT a domain transformer augmenting an abstract domain with logical connectors. This abstract domain would resemble the DPLL(T) algorithm of SMT solvers.

VIII. Conclusion

We believe that abstract domains are largely orthogonal to existing approaches. By encapsulating a constraint solver into the abstract domain PP, we can reuse it in a larger framework, and make it cooperate with other solvers, in our case octagon from abstract interpretation. In addition, we integrated reified constraints into our abstract solving framework by formalizing deduced constraints as under-approximations. This allows us to preserve correctness of the solving algorithm (we do not lose solutions). We have illustrated our approach on the RCPSP problem, but we should stress that our approach is generalist since PP can treat any constraint problem. Although our solver is only a prototype, the experiments show that our approach is efficient, especially to prove unsatisfiability. The next step is to formalize conflicts resolution (such as in SAT and SMT solvers) and to integrate it in the abstract solving framework.

References

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical computer science*, 221(1-2):179–210, 1999.

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[3] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints*, 18(1):1–6, January 2013.

[4] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2nd Edition (revision a), February 2011. SICS research report T2012-03, http://soda.swedish-ict.se/5195/.

[5] Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of finite domain constraints. *ICLP'94, International Conference on Logic Programming*, 1994.

[6] Aziem Chawdhary, Ed Robbins, and Andy King. Incrementally closing octagons. *Formal Methods in System Design*, January 2018.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[8] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM (JACM)*, 59(6):31, 2012.

[9] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[10] Vijay D'Silva and Caterina Urban. Abstract interpretation as automated deduction. *Journal of automated reasoning*, 58(3):363–390, 2017.

[11] Antonio J. Fernández and Patricia M. Hill. An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems*, 26(1):1–46, January 2004.

[12] Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 226–235. ACM, 2008.

[13] Birger Franck, Klaus Neumann, and Christoph Schwindt. Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling. *OR-Spektrum*, 23(3):297–324, 2001.

[14] S. Ilog. Revising hull and box consistency. In *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*, page 230. MIT press, 1999.

[15] Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. Benchmark instances for project scheduling problems. In *Project scheduling*, pages 197–212. Springer, 1999.

[16] Rainer Kolisch and Arno Sprecher. PSPLIB-a project scheduling problem library. *European journal of operational research*, 96(1):205–216, 1997.

[17] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006.

[18] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 4(3–4):120–372, 2017.

[19] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[20] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation*, pages 434–454. Springer, 2013.

[21] Vaughan Pratt. Two easy theories whose combination is hard. Technical report, Technical report, Massachusetts Institute of Technology, 1977.

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

[23] Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, December 2008.

[24] Christian Schulte, Guido Tack, and Mikael Lagerkvist. *Modeling and Programming with Gecode*, 2014.

[25] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *International Conference on Principles and Practice of Constraint Programming*, pages 746–761. Springer, 2009.

[26] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[27] Gagandeep Singh, Markus Püschel, and Martin Vechev. Making numerical program analysis fast. pages 303–313. ACM Press, 2015.

[28] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.

[29] Charlotte Truchet and Gérard Assayag. *Constraint Programming in Music*. Wiley, 2011.

[30] Petr Vilím. *Global constraints in scheduling*. PhD thesis, Charles University in Prague, 2007.