Abstract domains for constraint programming, with the example of octagons

Charlotte Truchet, Marie Pelleau, Frédéric Benhamou
Laboratoire d'Informatique de Nantes-Atlantique (LINA), UMR 6241
Nantes University, CNRS
2 rue de la Houssinière, Nantes, France
Email: firstName.lastName@univ-nantes.fr

***Abstract*—In Constraint Programming (CP), the central notion of consistency can be defined as a fixpoint of some contracting operators. These operators always deal with cartesian products of domains of the same nature (real intervals, integer sets, etc), due to the cartesian nature of the CSP format. However, *inside* the solving process, there is no particular reason why the domains should be cartesian. In another research field, Abstract Interpretation (AI) in semantics relies on a strong and elegant theory dealing with over-approximations of variables. It allows in particular to mix abstract domains of different kinds (integer, reals...). Several numerical abstract domains for continuous variables have recently been proposed, some of them cartesian, other relational. In this article, we adapt to CP the AI definition of abstract domains. We give an abstract consistency definition, and show it extends the usual CP consistencies. We also give a general solving algorithm for abstract domains. Finally, we propose the octagon abstract domain and study its practical feasibility.**

I. Introduction

Since the early work by Montanari [1], Constraint Programming (CP) provides many efficient algorithms to solve declarative problems over discrete or continuous variables. But the methods used in each case cannot be easily combined, due to a significant difference between the domains representations: integers can be enumerated, and reals cannot. They are even not representable on a computer and continuous CP techniques rely on interval arithmetics as defined in [2]. The intervals' lack of precision may then generate overestimations of the solutions (*e.g.,* when there is several occurrences of the same variable, when the constraints are not linear, ...). Several methods have been proposed to reduce this overestimation using estimators for nonlinear constraints [3], safe relaxation of quadratic constraints [4], or more recently the monotonicity of the constraints [5].

Also, real-life problems are often mixed: they have both integers and real variables. A good example is the problem of computer vision as described in [6]. Robot localization is another mixed problem that recently required the definition of a mixed global constraint [7]. But in general, mixed CP solving is still a challenge.

Instead of trying to improve the interval-based methods we decided to investigate the possible use in CP of *abstract domains* of Abstract Interpretation (AI) ([8]). These abstract domains have two features that seem interesting in CP. First, they can naturally deal with several type of domains (integers, floating points, etc). Mappings between different domain types have already been introduced in AI, both in theory and in practice. Second, recent works [9] showed the need for relational numerical abstract domains, which are more precise than the classical intervals. Our idea is to adapt the abstract domains of AI to the needs of CP, which gives a slightly different and much simpler definition. This can both improve the efficiency of the solving process and provide a good theory to combine the different solving methods for discrete and continuous variables. We present more precisely the case of the octagon abstract domains as an example. The goal of this work is threefold: study in general the links between CP and AI, use more powerfull abstract domains to improve the contraction operators and solving process, and possibly extend the CP theory to naturally deal with variables of different kinds.

The next two subsections aim at presenting the basic notions from CP and from AI that we will need afterwards. This presentation is not exhaustive. For a more complete presentation, we refer the reader to Barták's article [10] and Miné's PhD [9] respectively. Section II gives a definition for abstract domains adapted to CP needs. Section III details an example of abstract domains: the octagons.

A. Constraint Programming

We give here the basic definitions of some CP notions. They usually vary for discrete and continuous constraint problems (over integer or real variables). We write $\times$ for the cartesian product. Note that in all the article we will keep the same notations as introduced here.

Definition Given integers n and p, n sets $D_1...D_n$, a first-order logical language $\mathcal{L}$, and p atomic formulas $C_1...C_p$ of $\mathcal{L}$ (without quantifiers), a *Constraint Satisfaction Problem* (CSP) is given by:

- n variables $\mathcal{V} = (V_1...V_n)$,
- n domains $\mathcal{D} = (D_1...D_n)$,
- p constraints $C_1...C_p$.

Let $v^1...v^n$ be a sequence such that $v^i \in D_i$. It is a *solution* of the CSP iff $\forall j \in \{1...p\} C_j(v^1...v^n)$. The *solution set* $\mathcal{S}$ is the set of all solutions.

A *complete approximation* (resp. *sound approximation*), of the solution set is a sequence $w^1...w^n$ such that $w^i \subset D_i$, and $\mathcal{S} \subset w^1 \times ... \times w^n$ (resp. $w^1 \times ... \times wn \subset \mathcal{S}$).

The logical language $\mathcal{L}$ is just a tool to define constraints. It usually includes the usual function symbols (such as $+$, $-$ etc) and relation symbols (such as $=$, $\leq$ etc). Other relation symbols are often introduced in order to define global constraints. For example, alldifferent is a n-ary constraint and its semantic is: $\forall i \neq j, V_i \neq V_j$.

We have deliberately kept the domains' definition as vague as possible. Most of the time, they are finite subsets of the integers (discrete CSPs). But real variables can also be used. In that case, domains are usually subintervals of $\mathbb{R}$ with floating points bounds. Let $\mathbb{F}$ be the set of floating point numbers, according to the norm IEEE 754 [11] for instance. Let $\mathcal{I} = \{[a, b] \subset \mathbb{R}, a, b \in \mathbb{F}\}$. CSPs with domains in $\mathcal{I}$ are called continuous CSPs. In that case, a constraint solver does not provide solutions, but intervals containing the solutions.

By this definition, the number of all possible assignments of domains' values to the variables grows exponentially in n. Eventhough it is possible to define trivial CSPs, the CSPs of interest are NP-hard in general, at least in the discrete case. The range of academic and industrial applications is very large, from the n-queens to scheduling problems, frequency assignments, graph partitioning, etc [1].

The notion of *consistency* is central in CP. Several variants have been proposed, depending on the type of the CSP and other parameters. We only define the most usual ones. See [12] by Apt for a more complete description, as well as a theoretical framework to compare consistencies.

Definition Let $V_1...V_n$ be variables, over finite discrete domains $D_1...D_n$, and C a constraint.
The domains $D_1...D_n$ are said *generalized arc-consistent* (GAC) for C iff $\forall i \in \{1...n\}, \forall v \in D_i, \forall j \neq i, \exists v^j \in D_j$ s.t. $C(v^1, v^2, ..., v^{i-1}, v, v^{i+1}, ...v^n)$.
The domains are said *bound-consistent* (BC) for C iff $\forall i \in \{1...n\}$, D_i is an integer interval, $D_i = \{a_i...b_i\}$ for some $a_i, b_i \in \mathbb{N}$, and the previous condition applies to a_i and b_i: $\forall j \neq i, \exists v^j \in D_j$ s.t. $C(v^1, v^2, ..., v^{i-1}, v, v^{i+1}, ...v^n)$.

Intuitively, generalized arc-consistency is easier defined by its negation: a value of a domain that cannot satisfy the constraint (whatever the values for the other variables) is not consistent. Bound-consistency is the same notion when enforcing the domains to be integer intervals. In that case, the values which are strictly inside the interval are not considered. Both definition generalize in a natural way to several constraints.

Consistencies for continuous problems follow the same idea, but they are a little more difficult to define due to the fact that the real are not representable on a computer. Real domains are embedded in floating-points intervals. This leads to the following definition.

Definition Let $V_1...V_n$ be variables, over continuous domains represented by intervals $D_1...D_n \in \mathcal{I}$, and C a constraint.The domains $D_1...D_n$ are said *Hull-consistent* iff $D_1 \times ... \times D_n$ is the smallest floating-point box containing the solutions.

[1]Many detailed examples, as well as an encyclopedy of global constraints, can be found in http://www.emn.fr/x-info/sdemasse/gccat/

```
while termination criterion not met do
  while consistency has not been reached for all constraints do
    apply a constraint propagator to the domains
  end while
  if a domain is empty or termination criterion met then
    backtrack to the last backtrackable point
  else
    cut a variables' domain, mark as backtrackable point
  end if
end while
```

Fig. 1. General scheme for solving a CSP

Hull consistency can be computed by examining the syntax tree of the constraint, using Moore interval arithmetics (as defined in [2]) to reduce the domains. An efficient algorithms, HC4, has be given in [13]. The efficiency strongly depends on the multiple occurrences of variables, and recent improvements have been proposed in [14].

Any operator that acts on a CSP and achieves consistency for a constraint is called a propagator. The consistent domain for a particular CSP is computed by iterative applications of the constraints propagators until a fixpoint is reached. Propagators should decrease the domains w.r.t. inclusion, in order to guarantee the termination of the solving process, although recent work from Schulte and Tack [15] shows that this condition could be weakened.

Once a CSP is defined, one usually wants to solve it, that is, give the solutions (discrete problems) or approximations of the solutions at a given precision (continuous problems). This is not an easy task because of CP's combinatorial nature. The basic principle is to reduce the search space iteratively. Achieving consistency is a way to reduce the search space without losing potential solutions (this directly derives from the definition). But it is not sufficient in general: consistent domains may still contain non-solutions. The solving process is based on a branch and cut scheme, as summarized on figure 1. Although the general scheme is similar for discrete and continuous problems, the practical implementation differs significantly. Discrete domains are enumerized and in that case, "cutting" a domain means instanciate a variable. Continuous domains are iteratively cut into smaller parts.

The choice points (when a domain is cut) are marked so as to backtrack there if necessary. Backtracks are made either when the constraints are proven false (empty domain), or when it is no longer possible to continue (domains below a given precision if intervals, all variables instanciated if discrete). If a backtrack occurs, the solver restores the situation before the choice, and makes another choice. Termination criterion can be: one solution found, all the solution founds, a fixed precision has been reached, etc. One can imagine that the solver develops a *search tree* where the nodes are the backtrackable points and the sons of a node are the possible choices. This tree is built depth-first.

It is worth noticing that the solving process provides solutions in the discrete case, not in the continuous case. When the domains are intervals, the final answer is a cartesian product

of intervals, called a *box*, or a set of boxes, containing the solutions.

Under some hypothesis on the coherence between the constraint propagators and the cutting scheme, on can show that the solving process is sound and complete as done in [15].

B. Abstract Interpretation

It is a well known fact that the correctness of programs cannot be generically proven. To automatically analyze a code and prevent the bugs at compiling time (static analysis), AI relies on approximate semantics, enclosing the program's possible traces into a bigger semantic than the sound one. If the intersection between the approximated semantic and some dangerous zones (overflows for instance) is empty, we know that the program has no bugs. Of course, false positives can occur, in other words the process can raise alarms for bugs that are not.

The main principle is to focus on the values taken by the variables. These values change along the program, and form a trace that is called *concrete semantic*. They are replaced by an over-approximation of the possible values, called *abstract domains*. For every basic instruction of the program (*e.g.,* addition, etc), the abstract domains are modified accordingly. Of course, loop instructions are a particular case: there exists an operation allowing one to compute a fixpoint without having to analyze each iteration of the loop. The computed fixpoint is not the smallest one because it would need too much computation time, with regard to the number of lines to analyse, to reach it. A fixpoint that is conserved by the loop construction is sufficient for AI purposes. It is worth mentionning that in particular cases, AI allows to iterate the loop process, which is called local iterations [16] and is closer to the consistency computation in CP.

The theoretical tool behind this is the notions of *lattice*:

Definition A relation $\sqsubseteq$ on a non-empty set E is a *partial order* (po) iff it is reflexive, antisymetric and transitive. A partial order $(E, \sqsubseteq)$ is a *lattice* iff for $a, b \in E$, the pair $\{a, b\}$ has both a least upper bound and a greatest lower bound. It is *complete* iff any set has a least upper bound.

A complete lattice has a least element and a greatest element. Abstract domains are imposed to be lattices or complete partial orders, so that the iterative chains defined by the loop construction converge. They also come with operators allowing one to compute efficiently the loop fixpoints (at the price of losing precision).

Another important feature of AI is that abstract domains are linked by Galois connections:

Definition Given two abstract domains D_1 and D_2 (each being a po), a Galois connection is defined by two morphisms $\alpha : D_1 \to D_2$ and $\gamma : D_2 \to D_1$ such that

- α and γ are monotonic,
- $\forall X_1 \in D_1, X_2 \in D_2, \alpha(X_1) \sqsubseteq X_2 \iff X_1 \sqsubseteq \gamma(X_2)$.

Notice that $(\alpha \circ \gamma)(X_2) \sqsubseteq X_2$ and $X_1 \sqsubseteq (\gamma \circ \alpha)(X_1)$: abstract domains do not lose elements.

Abstract domains of various shapes have been defined, and they can involve one variable (non-relational domains) or several variables (relational domains). An abstract domain comes with operators allowing to make computations on them. For a variable set $\mathcal{V}$, a numerical abstract domain $D^\sharp$ is given by:

- a set $\mathcal{D}^\sharp$ whose elements are computer-representable,
- a partial order on $\mathcal{D}^\sharp$, with effective algorithms to compute it,
- smallest and greatest elements,
- a partial Galois connection with concrete domains,
- effective algorithms to compute several operators: sound abstractions for operators $\cup$, $\cap$, etc, narrowing and widening if necessary.

Abstract domains can be non-relational, which means that they are cartesian products of abstract domains for each variable. An example of non-relational abstract domain is the intervals $\mathcal{I}$ as introduced above. The partial order is given by inclusion. Concretisation is defined by $\gamma([a, b]) = \{x \in \mathbb{F}, a \leq x \leq b\}$: note that only floating points are allowed, because the concrete semantic is defined on floating point numbers. Abstraction is defined in a natural way by the interval including all the concrete values. The other operators, as well as complete definition, can be found in [17].

They can also be relational, and involve several variables. For instance, the polyhedron abstract domain is relational, and allows to take into account linear relations between the variables. So are the octagon abstract domain (restriction of the polyhedron) [18], or the interval polyhedron which extends polyhedron [19]. The reason why there are so many abstract domains is the trade-off between precision and computation time. The choice of an abstract domain depends on the property to be analyzed.

II. Abstract domains in Constraint Programming

Let us first discuss the links between AI and CP. From far, there are many similarities. On each side, the goal is to compute an over-approximation of the set of interest: traces or solutions. The underlying theoretical structures are also very similar: both work on lattices (abstract domains or variables' domains) and compute fixpoints.

But looking closer, an important practical point also differ: the precision of the over-approximations. AI uses over-approximations to prove that the traces do not intersect dangerous places (overflows, divisions by zero for instance). The precision must just be good enough to separate the traces from these places. In practice, the fixpoints for loop are computed in the abstract domain, in one shot, by specific operators (except when local iterations are performed) and overestimated. The gain in computation time allows to analyze very big programs. On the contrary, CP uses over-approximation in order to cut the search space, and they need to be as precise as possible. The fixpoints are computed in the concrete domain (the actual variables domains) by iterating propagators until the domains are stable.

Our idea is to benefit from the common points to bring back to CP a particular strength of the AI theory: it copes naturally and in an elegant way with abstract domains of various kind, while CP restricts consistency definitions to cartesian products of the domains, and these definitions are *ad hoc* for discrete and continuous problems. Our goal in particular is to study the application of numerical abstract domains to CP, in particular the ones defined by Miné in [9], [18], [19].

A. Abstract consistency

We propose here a new definition for consistency that: includes the most classical ones (Box, Hull, GAC, BC), and allows any kind of underlying structure. Note that a similar idea has been proposed in [20], focusing on a framework in which solvers of different type (discrete, continuous) can collaborate. Our work somehow extends their, but we place the domains shape at the center of the theory.

In all the following, we consider a CSP $(\mathcal{V}, \mathcal{D}, C_1...C_p)$. Let $(E, \subset)$ be a partial order for inclusion (elements of E are sets).

Definition (and proposition) Consider a constraint C of the CSP and $\mathcal{S}_C$ the solutions of C in $\mathcal{D}$.. An element e is E-consistent for C iff it is a least element for $\{e' \in E, \mathcal{S}_C \subset e'\}$. If E is closed by intersection, it is a complete lattice, and then there exists a unique E-consistent element for C, which is in E. In that case, it is written $\mathcal{C}_{E,C}$.

Proof: Let $\mathcal{C}_{E,C} = \cap_{e \subset E, \mathcal{S}_C \subset e} e$. Existence and unicity come directly from the fact that E is closed by intersection. ■

Definition Let $e \in E$. It is E-consistent for $C_1...C_p$ (or any subset of these) iff it is a least element of $\{e' \in E, \mathcal{S} \subset e'\}$. If such an element exist, it will be written $\mathcal{C}_{E,C_1 \wedge ... \wedge C_p}$ or $\mathcal{C}_E$ if there is no ambiguity.

Proposition 2.1: If E is closed by intersection, it is a complete lattice, and then $\mathcal{C}_E$ exists and is unique. In addition, the set of all $\mathcal{C}_{E,C_{i_1} \wedge ... \wedge C_{i_k}}$ for $i_1...i_k \in \{1...p\}$ form a lattice for inclusion and $\mathcal{C}_{E,C_1...C_p}$ is its least element.

Proof: We first prove the lattice structure. Of course, inclusion is already a partial order. Let $i, j \in \{1..p\}$ and consider $\mathcal{C}_{E,C_i}$ and $\mathcal{C}_{E,C_j}$. Obviously, $\mathcal{C}_{E,C_i \wedge C_j} \subset \mathcal{C}_{E,C_i} \cap \mathcal{C}_{E,C_j}$ and provides a least element for $\mathcal{C}_{E,C_i}$, $\mathcal{C}_{E,C_j}$. On the other side, $\cap_{e \in E, \mathcal{C}_{E,C_i} \subset e, \mathcal{C}_{E,C_j} \subset e} e$ is a greatest element. Thus the set of all $\mathcal{C}_{E,C_{i_1} \wedge ... \wedge C_{i_k}}$ is a lattice. It is complete because finite. By the same remark as above, $\mathcal{C}_{E,C_1...C_p}$ is the least element. ■

The proposition is rather formal but still very usefull: provided that each constraint of the CSP comes with a propagator, it is sufficient to apply these propagators recursively until the fixpoint is reached, no matter the order, to achieve consistency for the CSP. On existing consistencies, a similar idea has been proposed by Apt in [12].

B. Retrieving existing consistencies

This part is merely technical, however it ensures that the consistencies defined above can actually be derived by our definition.

Proposition 2.2: Let $E_1 = \{d_1 \times ... \times d_n, d_i \subset D_i\}$. The E_1-consistency is generalized arc-consistency.

Proof: We first prove that GAC $\implies$ E_1-consistent. Let $D = D_1 \times ... \times D_n$ GAC for a constraint C. It obviously contains all the solutions, and we need to prove that it is the smallest such element of E_1. Let $D' \in E_1$, strictly smaller than D. Then there is a i s.t. $D'_i \subsetneq D_i$ and a $v \in D_i \backslash D'_i$. Since $v \in D_i$ and D is GAC, there also exist $v^j \in D_j$ for $j \neq i$ s.t. $(v^1...v...v^n)$ is a solution. This solution is not in D' and thus all $D' \subsetneq D$ lose solutions.
We now prove that E_1-consistent $\implies$ GAC. Let $D = D_1 \times ... \times D_n$ E_1-consistent for a constraint C. Let $i \in \{1...n\}$ and $v \in D_i$. Suppose that for all other $v^j \in D_j$, $(v^1...v...v^n)$ is not a solution: we can construct the set $D_1 \times ... \times (D_i \backslash \{v\}) \times ...D_n$ which is strictly smaller that D and also contains all the solutions. Hence there can be no such i and v, and D is GAC. ■

Proposition 2.3: Let $E_2 = \{\{\underline{d_1}...\overline{d_i}\} \times ... \times \{\underline{d_n}...\overline{d_n}\}, \underline{d_i}, \overline{d_i} \in D_i\}$. The E_2-consistency is bound-consistency.

Proof: Easy by adapting the previous one. ■

Proposition 2.4: The $\mathcal{I}$-consistency is hull-consistency.

Proof: Obvious from the definition. ■

C. CP abstract domains

We propose a definition for abstract domains in CP, in the same spirit as AI's abstract domains. The main difference lies in the fact that we do not need the AI operators. But we add a splitting operator, which allows to model the cutting scheme in the solving process.

Definition Let $(E, \subset)$ be a complete poset. A *splitting operator* is an operator $\oplus : E \to \mathcal{P}(E)$ such that $\forall e \in E$,

- $|\oplus(e)|$ is finite, $\oplus(e) = \{e_1...e_k\}$,
- $\cup_{1 \leq j \leq k} e_j = e$,
- $\forall j \in \{1...k\}$, $e \neq \emptyset \implies e_j \neq \emptyset$,
- $e_j = e \implies e$ is a least element of $E \backslash \emptyset$.

The first condition is needed for the search process to terminate (finite width of the search tree). The second condition enforces that no parts of the search space will be left outside the solving process. We could have asked $\oplus(e)$ to be a partition of e, but this would have been a little complicated to adapt to the continuous case (because of the borders) and it is not mandatory to prove completeness of the solving process. One can imagine a splitting operator that gives twice the same values. The solving process will be inefficient, but still complete. The third condition is merely technical, to avoid useless checks, and keep the fact that empty domains characterize inconsistency. The fourth condition means that the splitting operator actually splits: it is forbidden to stay on the same domains.

Example For $E_1 = \{d_1 \times ... \times d_n, d_i \subset \mathbb{N}\}$, one can define for every i in $\{1...n\}$:
$\oplus_{E_1,i}(d_1 \times ... \times d_n) = \{d_1 \times ... \times d_{i-1} \times \{v\} \times d_{i+1} \times ... \times d_n, v \in d_i\}$. This consists in choosing a variable V_i and a value v in D_i. It models the instanciation scheme for discrete CSPs.

Example For $E_3 = \mathcal{I}^n$, for a $I_1 \times ... \times I_n \in E_3$, and a $i \in \{1...n\}$, one can define $\oplus_{E_3,i}(I_1 \times ...I_n) = \{I_1 \times ... \times I_{i-1} \times I_i^{\vdash} \times ... \times I_n, I_1 \times ... \times I_{i-1} \times I_i^{\dashv} \times ... \times I_n\}$, where $I^{\vdash}$ is the lower half of I and $I^{\dashv}$ its upper half (with whatever way of managing the rounding errors on floats, provided $I^{\vdash} \cup I^{\dashv} = I$). This consists in cutting I_i into two parts and models the traditional splitting process for continuous CSPs.

We now define CP abstract domains. We borrow from AI the definition for concretisation and impose a minimal Galois connection with at least a cartesian abstract domain E_1 or E_3.

Definition An *abstract domain* $D^\sharp$ is given by :

- a complete lattice E,
- a sequence of splitting operators on E,
- a concretisation $\gamma : D^\sharp \to D^\flat$, and an abstraction $\alpha : D^\flat \to D^\sharp$ that form a Galois connection *with a non-relational domain*,
- a strictly decreasing function $\rho : D^\sharp \to \mathbb{R}$.

Concretisation allows to interpret the abstract domains, concrete domain $D^\flat$ are the domains of the CSP semantics (intervals or integer sets). The Galois connection links the abstract domains to the computer representation of the variables of the CSP. CP aims at providing solutions to a CSP, which is cartesian by definition. The concrete domains should thus be cartesian. Whatever domains are used inside the solver, we need to provide a cartesian answer, that is, to link every variable with a domain representing the solutions. If the abstract domains is itself cartesian, this can simply be identity. If they are not, abstraction allows to initially compute the abstract domains from $\mathcal{D}^\flat$ and concretisation to project the abstract domains on the variables when the solving process is finished. Note that such Galois connection are given for Miné's abstract domains, and we can simply borrow them.

The function ρ is intended to represent some measure on the abstract domains, measuring their precision. This is mandatory to express the termination criterion, especially for continuous domains.

Example For a fixed n, the set E_1 with the splitting operators $\oplus_{E_1,i}$ for $i \in \{1...n\}$ is an abstract domain, closed by intersection. In order to model the fact that the search process ends when all the variables are instanciated, one can take $\rho_1(e) = max(|D_i|, i \in \{1...n\})$ and stop when ρ_1 is equal to 1.
E_1 is cartesian and well suited for discrete problems. As mentioned earlier, the E_1-consistency is GAC.

Example For a fixed n, the set E_3 with the splitting operators $\oplus_{E_3,i}$ for $i \in \{1...n\}$ is an abstract domain, closed by intersection. In order to model the fact that the search process ends when a given precision r is reached, one can take $\rho_3(e) = max(\overline{d_i} - \underline{d_i}, \text{for} D_i = [\underline{d_i}, \overline{d_i}], i \in \{1...n\})$ and stop when $\rho_3 \leq r$.
E_3 is cartesian and well suited for continuous problems. As mentioned earlier, the E_3-consistency is Hull consistency.

int $j \leftarrow 0$ {*j indicates the depth in the search tree*}
int op[] {*at a depth j, stores the index of the chosen splitting operator, initialized uniformly at* 0}
int width[] {*at a depth j, stores the width of the tree already explored, initialized uniformly at* 0}
while $\rho(D) > r$ **do**
  apply E-consistency
  if a domain is empty **then**
    backtrack to the last backtrack point (and restore j)
  else
    if width[j]==0 **then**
      op[j] $\leftarrow i$ {*choose a splitting operator*}
    end if
    D $\leftarrow \oplus_{E,i}(D)_{\text{width}[j]}$
    width[j]++
    j++
  end if
end while

Fig. 2. Solving with abstract domains. The set of splitting operators is $\{\oplus_{E,i}, 1 \leq i \leq k\}$. The process stops when a precision r has been reached. Everytime a choice is made on the partition obtained from the application of a splitting operator, the point is marked as backtrackable.

Abstract domains can be defined independently from the domains of a particular CSP. They are intended as shapes for the consistency computations. Of course, they can be cartesian, but this is not mandatory. Note that we do not include the propagators, because they are dedicated to particular constraints.

D. Solver

It is now straightforward to build the general skeletton of a solver. A solver alternates propagator applications and cuts, in the same way as on figure 1, but the whole process is now parametrized by the type of abstract domain which is chosen, as shown on figure 2. For coherence reasons, we impose that, at a given depth j in the search tree, the splitting operator is the same throughout the backtracks. The first time we reach depth j, a splitting operator $\oplus_i$ is thus chosen once and for all. It is applied to the current domain D which gives $\oplus_i(D) = \{D'_1...D'_k\}$. Then, when backtracks occur, the cut domains D'_l are visited for l from 1 to k.

But one can choose any kind of splitting operator from one step to the other. This is part of the search strategy: choose the biggest domain and cut it in two equal parts for continuous CSP, choose the smallest domain for discrete CSP, etc.

With some light hypothesis on the propagators (monotonicity: the propagators do not increase the domains), we assume that this terminates and is complete. The proof can be adapted from the one given in [15]. For termination, the skeletton of the proof is to look at one branch on the search tree, use the fact that the sequence of abstract domains encountered on this branch is strictly decreasing (due to the definition of splitting operators), and the complete lattice structure of the abstract domain. Completeness is proven by induction on the search tree, by the consistency definition (propagators do not lose solutions) and the fact that splitting operators cover all the search space.

III. Example of octagons

In this section we focus on continuous constraints on real variables, and define an octagonal hull consistency. The idea is that octagons, or other relational domains, may improve the precision of the over-approximations of the solutions, which may improve the solving process at several levels. First, octagon consistency is by nature more precise than interval consistency, for a cost that is not so high. Second, an octagon can prove inconsistency more easily than intervals. Last but not least, octagon-based pruning should give less boxes than intervals (because the boxes are somehow more expressive). We then describe a very simple prototype implementation and discuss the results.

Given a set of variables $\mathcal{V} = \{V_1...V_n\}$, Miné calls *octagonal constraints* any constraint of the form $\pm V_i \pm V_j \leq c$. Note that this includes constraints of the form $V_i \leq c$ (choose $i = j$). The octagons Oct are defined as a set of octagonal constraints, and they have at most $2n^2$ sides in dimension n. In dimension 2, an example is given on the left hand side of figure 3. Concretisation γ^{Oct} is defined as the set of floating points satisfying the system.

A. Octagons abstract domains

We define the octagons exactly the same way, except that concretisation will be given by the set of *real* points satisfying the constraints: if m is an octagon, $\gamma^{Oct}(m) = \{(v^1...v^n) \in \mathbb{R}^n, (v^1...v^n)$ satisfy the constraints of $m\}$.

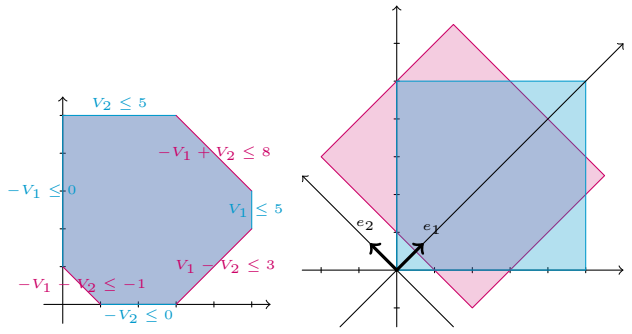


Fig. 3. Example of an octagon, with its representation in Oct on the left, and its representation as the intersection of two boxes on the right.

Note that Miné uses a particular structure (Difference Bound Matrix) to represent the octagons and make computations on them. They also show that the DBM representation of octagon forms a complete lattice. We rely on the classical interval representation of continuous CSPs to give a rather different and simpler implementation, which is sufficient for our needs. Our representation is based on the intersection of rotated boxes. Note that the idea of rotating the basis has already been introduced by Goldsztejn et al. in [21]. Their goal was to adapt the basis to the shape of the solution set.

Call $\mathcal{E} = (e_1...e_n)$ the canonical basis of $\mathbb{F}^n$, and for $i, j \in \{1...n\}$, $i < j$, let $\mathcal{E}^{i,j} = (e_1...e_{i-1}, \frac{1}{\sqrt{2}}(e_i + e_j), e_{i+1}...e_{j-1}, \frac{1}{\sqrt{2}}(-e_i + e_j), e_{j+1}...e_n)$, the basis obtained by rotating e_i and e_j by $\frac{\pi}{4}$. An octagon will be represented by the intersection of boxes, each of them in a particular $\mathcal{E}^{i,j}$. The first box, called canonical box, is initially given by $B^{Can} = D_1 \times ... \times D_n$. The other boxes $B^{Rot}_{i,j}$, called rotated boxes, live in the other basis $\mathcal{E}^{i,j}$. The resulting octagon is the intersection of these boxes (but in fact, we will not really need to compute this intersection effectively). Provided we have an order for the different rotated basis, an octagon can simply be stored as $(B^{Can}, B^{Rot}_{1,2}...B^{Rot}_{1,n}...B^{Rot}_{n-1,n})$. Abstraction and concretisation with the interval abstract domains have already been defined by Miné in [9].

We then propose a family of splitting operators on Oct. The idea is to cut the octagon into two equal parts on one axis, similarly to the interval abstract domains. To define a splitting operator $\oplus^{i,j}_k$, we choose a basis $\mathcal{E}^{i,j}$, an axis k and, for each box $B^{Rot}_{i,j} = I_1 \times ... \times I_n$, we generate $\{I_1 \times ... \times I_{k-1} \times I_k^{\vdash} \times ... \times I_n, I_1 \times ... \times I_{k-1} \times I_k^{\dashv} \times ... \times I_n\}$ (we simplify the notations by forgetting ^{Rot} and $_{i,j}$ when no confusion can occur). It is easily checked that this is indeed a splitting operator.

We define also the precision measure on octagons as $\rho^{Oct}(B^{Can}, B^{Rot}_{1,2}...B^{Rot}_{1,n}...B^{Rot}_{n-1,n}) = \rho_3(B^{Can})$. We do not consider the rotated basis because we do want the final box obtain by the Galois connection to be at the given precision. However, this would not change much.

Remark The poset $(Oct, \subset)$ is closed by intersection, and forms a complete lattice.

B. Effective computation of octagon consistency

We now show how to compute octagonal consistency, by reusing existing algorithms, such as HC4 introduced by [13].

The first step is to adapt the constraints to the rotated basis. This requires symbolic computation, but is very simple. For basis $B^{Rot}_{i,j}$, we just need to replace every occurrence of V_i by $\frac{1}{\sqrt{2}}(V_i + V_j)$ and V_j by $\frac{1}{\sqrt{2}}(-V_i + V_j)$. Doing this may create new multiple occurrences of variables, which are a problem for HC4. We shall factorize the new expression whenever it is possible.

We now have expressions of the constraints in each basis (although they still have the same semantic). Each constraint has its own hull-consistent box, that can be computed using HC4. By the complete lattice structure, the consistent octagon is unique and can be computed by applying intersection and HC4 on these boxes until the fixpoint is reached (no matter the order). This property allows to adapt the HC4 algorithm to octagons.

As shown on figure 4, we can imagine two strategies. The first one, in dashed lines, applies the Hull consistency in parallel in each basis, then intersects the consistent boxes to obtain an octagon. As the intersection can reduce the boxes, these two steps are repeated until a fixpoint is reached.

In the second version, in solid line on the figure 4, we fix a constraint, say C_1, and compute the hull-consistent box in each basis. These boxes are intersected and the C_1 consistent octagon is the starting point for the next constraint to propagate. This is repeated until a fixpoint is reached. Intuitively, this corresponds to the true octagonal consistency.

Notice that the intersection can be done in linear time, here the most expensive part in terms of complexity, is the HC4 part. In dimension 2, we use it twice as much as for the classical Hull consistency. However, for a problem in dimension n we can either apply the same strategy and apply n rotations, one for each axis and keep the octagonal structure; or apply only fewer rotations.

We have implemented a prototype of this, relying on a C++ library called Ibex, presented in [22]. We use the Ibex implementation of HC4. This implementation is still a prototype, and does not include all the constraint language yet. We used it to test the feasibility of octagonal consistency. We have tested octagonal consistency on problems from the Coconut benchmark[2] but adapted to our language (we suppress the optimization constraints). They are plotted on figure 6. We show the final concretised boxes, that is, $\gamma(m)$ if m is the final octagon, and compare these boxes to the ones obtained by mere interval propagation.

On the first example, 6(a), the Oct contraction does not improve the precision. This is due to the new multiple occurrences in the rotated constraints. On the contrary, in the second example 6(b), there exist multiple occurrences in the canonical basis, and none in the rotated basis. By factorizing the adaptation of the constraint in the rotated basis, we eliminate all the multiples occurrences, which allows us to optimally contract it in rotated basis, even if the projected box in $\mathcal{I}$ is not the smallest one. On the other examples, 6(c), 6(d), 6(e), the Oct contraction is optimal. In general, we can see that octagonal propagation improves the precision, as detailed also on Figure 5.

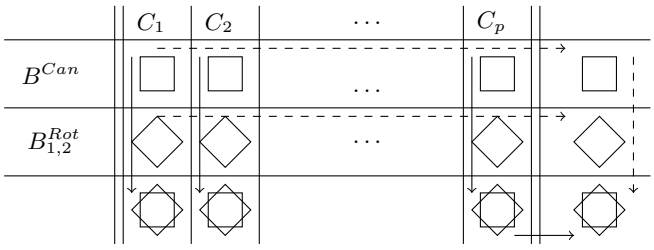


Fig. 4. Scheme of the possible heuristics for octagonal consistency. In dashed line the first version, where the propagations on the two CSPs are done in parallel. In solid line the second one, with octagonal consistency for each constraint.

C. Solving with octagons

We have also made a prototype implementation of an octagon solver. Again, it is based on the Ibex solver. For now, we do not have implemented a true octagon solving but used the Ibex solver in each basis and made them communicate when necessary. This is of course very inefficient in terms of computation time, but it allowed us to study some basic octagonal strategies.

The different basis have been coded as independent problems, each one having its own Ibex solver. We make them communicate when a split occurs: we dynamically post new constraints (linear inequalities) to the other solvers. The choice of the splitting operator is made by browsing all the solvers, in order to choose which domain to cut.

[2]The Coconut benchmark can be found at http://www.mat.univie.ac.at/~neum/glopt/coconut/

	Domain	$\mathcal{I}$ contraction	Oct contraction
a	$[-\infty,+\infty]^2$	$[-1,1]\times[0,1]$	$[-1,1]\times[0,1]$
booth	$[-\infty,+\infty]^2$	$[-\infty,+\infty]^2$	$[0,2]\times[2,4]$
circle	$[-5,5]^2$	$[2,3]\times[0.27,2]$	$[2,3]\times[0.38,2]$
hs8	$[-\infty,+\infty]^2$	$[-5,5]^2$	$[-5,5]^2$
triangle	$[-2,5]^2$	$[-2,5]\times[-2,2]$	$[-2,2]^2$

Fig. 5. Examples of octagonal consistency in dimension 2. The computation time is not given because it is negligible (one measures 0 seconds for every propagation).

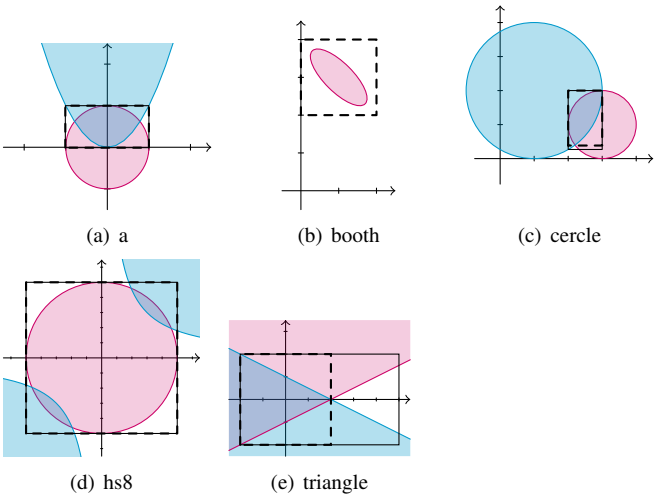
We have tested several splitting strategies. The first version applies the $\oplus_k^{i,j}$ for all $i,j,k \in \{1...n\}$, and posts the corresponding constraint in all the other basis (Pure Octagonal Version or POV). But, with our basic implementation, boxes are overlapping which implies that some computations are made twice, and the results cannot be interpretated. The second version applies only the $\oplus_k^{Can}$ for $k \in \{1...n\}$, that is, only the splitting operator for the canonical basis. In that case, we still post the corresponding constraints to the other basis, which may improve the borders (Normal Guided by Octagons or NGO). We also tried the Octagonal Contraction (OC) strategy, which simply applies octagonal consistency once, before starting to solve. In all cases, the termination condition is given by $r = 0.01$.

Figure 7 shows the results for the same problems as above, in dimension 2. The computation time is not so good with NGO, which we believe is due to our poor implementation of the octagonal splitting: the solver needs to maintain two solvers along with communications between them, which is in itself costly in time. But we can also see that the number of created boxes, as well as the number of boxes of the solution, is smaller with the NGO. Usually, interval solvers spend much more time in creating boxes by splitting than applying propagators. Hence these results are encouraging and show that it is worth implementing a real octagonal solver.

IV. Conclusion

We have shown the feasibility of using abstract domains, inspired from AI, in constraint solving. Our prototype implementation produces results which are bad in computation time, but good in terms of created boxes. The first conclusion is that it is now worth considering a real implementation of octagon solving to get rid of the problems induced by the management of several solvers. This should allow to extend the solving possibilities and try real octagonal search strategies (splitting strategy for instance). We also need to refine the termination condition, to avoid splitting fully satisfied or fully unsatisfied octagons.

There are many straightforward possible extensions of the octagon domain. A clear drawback of octagons is their high number of faces, which forces to maintain a high number of boxes (whatever the representation). It is easy to consider



(a) a (b) booth (c) cercle

(d) hs8 (e) triangle

Fig. 6. Examples of problems in dimension 2, with the interval consistency plotted in solid line and the octagonal consistency in dashed line.

	Interval version		OC		NGO	
a	*24684* 49367	1.2s	*24684* 49367	1.2s	*24625* 49249	5.4s
booth	*31380* 62809	1.5s	*28539* 57135	1.4s	*27476* 55105	5.9s
circle	*31399* 62797	1.6s	*31285* 62569	1.6s	*29708* 59415	6.7s
hs8	*99168* 198335	5.1s	*99168* 198335	5s	*95101* 190201	21.5s
triangle	*210665* 421329	10.2s	*247098* 494195	12.4s	*218575* 437149	48.5s

Fig. 7. Results of two octagonal strategies compared to the classical interval solver. The left column gives the number of boxes in a solution, in italic, and the total number of created boxes. The right column gives the computation time.

octagons with a smaller number of faces, as for instance zones [9]. In the same spirit, one could consider octahedrons as defined by Clarisó and Cortadella in [23], with a smaller number of rotated boxes. The work by Goldsztejn et al. in [21] provides another interesting possible development. They rotate the problems using a gradient method, to improve the fitness of the consistent box to the shape of the solution set. Following this idea, we could adapt the octagon to paralleloids (by simply changing the rotation angle) so that the consistent box is tighter, and closer to solution set.

In the end, the initial idea behind this work was to use the Galois connections of Abstract Interpretation to define solvers able to deal with both integer and real variables. We have given an abstract domains-based solving algorithm and described abstract domains for integer and real variables. This work is thus a first step towards a mixed constraint solver.

References

[1] U. Montanari, "Networks of constraints: Fundamental properties and applications to picture processing," *Information Sciences*, vol. 7, no. 2, pp. 95–132, 1974.

[2] R. E. Moore, *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[3] G. Borradaile and P. V. Hentenryck, "Safe and tight linear estimators for global optimization," *Mathematical Programming*, vol. 102, pp. 495 – 517, 2005.

[4] Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J. pierre Merlet, "Efficient and safe global constraints for handling numerical constraint systems," *SIAM Journal on Numerical Analysis*, vol. 42, no. 5, pp. 2076–2097, 2004.

[5] I. Araya, B. Neveu, and G. Trombettoni, "An interval constraint propagation algorithm exploiting monotonicity," in *Proceedings of the 4th International workshop on Interval Analysis, Constraint Propagation, Applications*, 2009.

[6] L. Jaulin and S. Bazeille, "Image shape extraction using interval methods," in *Proceedings of the 15th IFAC Symposium on System Identification*, 2009.

[7] G. Chabert, L. Jaulin, and X. Lorca, "A constraint on the number of distinct vectors with application to localization," in *CP'09: Proceedings of the 15th international conference on Principles and practice of constraint programming*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 196–210.

[8] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," in *Proceedings of an ACM conference on Language design for reliable software*. New York, NY, USA: ACM, 1977, pp. 77–94.

[9] A. Miné, "Weakly relational numerical abstract domains," Ph.D. dissertation, École Normale Supérieure, December 2004.

[10] R. Barták, "Foundations of constraint satisfaction," *IJCAI 2003*, 2003.

[11] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[12] K. R. Apt, "The essence of constraint propagation," *Theor. Comput. Sci.*, vol. 221, no. 1-2, pp. 179–210, 1999.

[13] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget, "Revisiting hull and box consistency," in *Proceedings of the 16th International Conference on Logic Programming*, 1999, pp. 230–244.

[14] I. Araya, B. Neveu, and G. Trombettoni, "A new monotonicity-based interval extension using occurences grouping," in *Proceedings of the 4th International workshop on Interval Analysis, Constraint Propagation, Applications*, 2009.

[15] C. Schulte and G. Tack, "Weakly monotonic propagators," in *Fifteenth International Conference on Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, I. Gent, Ed., vol. 5732. Lisbon, Portugal: Springer-Verlag, Sep. 2009, pp. 723–730. [Online]. Available: http://www.ict.kth.se/~cschulte/paper.html?id=SchulteTack:CP:2009

[16] P. Granger, "Improving the results of static analyses of programs by local decreasing iterations," in *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1992.

[17] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of ISOP'76*, 1976, pp. 106–130.

[18] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[19] L. Chen, A. Miné, J. Wang, and P. Cousot, "Interval polyhedra: An abstract domain to infer interval linear relationships," in *Proceedings of the 16th International Static Analysis Symposium (SAS'09)*, 2009, pp. 309–325.

[20] F. Benhamou, "Heterogeneous constraint solvings," in *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, 1996, pp. 62–76.

[21] A. Goldsztejn and L. Granvilliers, "A new framework for sharp and efficient resolution of ncsp with manifolds of solutions," in *Proceedings of the 14th international conference on Principles and Practice of Constraint Programming (CP '08)*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 190–204.

[22] G. Chabert and L. Jaulin, "Contractor programming," *Artif. Intell.*, vol. 173, no. 11, pp. 1079–1100, 2009.

[23] R. Clarisó and J. Cortadella, "The octahedron abstract domain," in *Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, 2004, pp. 312–327.