Joint NCC & IIS Bull., Comp. Science, 16 (2001), 1–27
© 2001 NCC Publisher

Towards a framework for designing constraint solvers and solver collaborations

C. Castro, E. Monfroy

In this paper, we propose a strategy language for designing constraint solvers and schemes of solver collaborations. Solvers are seen as bricks that can be integrated when creating more complex solvers that can become themselves new bricks to compose new solvers. These bricks are glued together using operators of our language. A pattern of operators can be used to create solvers and collaborations for solving different types of constraints. We illustrate the use of this language by describing some well-known techniques for propagation-based solvers, optimization problems, and collaboration of solvers (symbolic/numeric cooperation, linear/non-linear collaboration, local consistency combination).

1. Introduction

In the last twenty years, constraint programming has emerged as a new programming paradigm. In this alternative approach, the programming process is merely a specification of a set of requirements (i.e., the constraints), a solution for which will be generated using some general or domain specific techniques and mechanisms (i.e., the constraint solvers). Numerous algorithms have been developed for solving constraints and the resulting technology has been successfully applied to solving real-life problems.

The design and implementation of these constraint solvers is generally an expensive and tedious task, and thus, the idea of reusing solvers "of the shelves" is very interesting and promising [26]. However, that also implies that we must have some tools to integrate/combine them. Another key-point is that some problems cannot be tackled or efficiently solved with a single solver. Hence, we definitely realize the interest in integrating and making cooperate several solvers [16, 6, 19, 25, 23]. This is called collaboration of solvers [24]. In order to make solvers collaborate, the need for powerful strategy languages to control their integration and application has been well recognized [21, 22, 2].

However, the existing approaches are generally not generic enough: they consider fixed domains (linear constraints [6], non-linear constraints over real numbers [23, 19, 4]), fixed strategies, or fixed scheme of collaboration (sequential [23, 4], asynchronous [19]). In the language **BALI**, collaborations are specified using control primitives and the constraint system is a parameter. Although BALI is more generic and flexible, the control capabilities for specifying strategies are not always fine enough [22]: the control is based on a "set of constraints", not at the level of constraints taken separately. In the system COLETTE [11, 12], a solver is viewed as a strategy that specifies the order of application of elementary operations expressed by transformation rules. In this framework, different domains mainly mean the definition of different transformation rules, and different heuristics mean different strategies. However, the granularity of control is too low to really consider solver collaborations.

Extending the ideas of BALI and COLETTE, we consider collaborations of solvers as strategies that specify the order of application of solvers. In [9], we proposed a strategy language for designing elementary constraint solvers and we exemplify its use by specifying several solvers (such as solvers for constraints over finite domains and real numbers). In [10] we presented the application of our language to prototyping the constraint solving schemes via collaboration of solvers. In this paper, we show that designing solvers and collaborations are intrinsically linked and related. For example, the same strategy can be used to write a solver or to express a collaboration. In fact, the basic solvers are bricks that can be used to design more complex solvers and collaborations which then become other bricks. They can be re-used, assembled together through strategies, used in higher collaborations, ... The glue between these bricks (i.e., patterns of solvers and strategies, or assembly of operators) can be instantiated for different domains of constraints and different strategies of resolution. In this paper, these techniques are illustrated by numerous examples over different domains: generic propagation-based solvers (and instantiation for finite domains and real interval constraints), and collaboration of solvers (optimization problems, symbolic-numeric cooperation, linear/non-linear collaboration, local consistency combination). For each of these solvers/collaborations, we simulate the standard techniques, and we also propose improvements in terms of strategies.

The main motivation for this work is to propose a general framework in which one can design the component constraint solvers, as well as solver collaborations. This approach makes sense, since the design of constraint solvers and the design of collaborations require similar methods (strategies are often the same: *don't-care*, *fixed-point*, *iteration*, *parallel*, *concurrent*, ...). In other words, we propose a language for writing the component solvers and designing collaborations of several solvers at the same level. Key points in this work are the concepts of constraint filters, separators, and sorters. These notions allow one to manage constraints with high-level mechanisms. Furthermore, they help describing syntactical transformations and manipulations that are generally hidden in the implementation of the current solvers. These concepts are used to define the strategy operators for applying solvers, such as don't care mechanism, best application of solvers, concurrent solvers, parallel applications, and operators for treating sub-problems. These operators allow us to design solvers by combining the basic functions, as well as collaborations of solvers by combining the component solvers.

This paper is organized as follows: in Section 2, we use a very simple example to informally present our language. In Section 3, the basic definitions are given and we introduce the notions of filters and sorters. Section 4 details the basic and complex operators for applying solvers. In Section 5, we give a generic propagation-based solver together with some instantiations for finite domain constraints and interval constraints. Section 6 introduce optimization problems and some possible implementations using our language. Section 7 is devoted to different forms of solver collaborations: symbolic/numeric cooperation, linear/non-linear collaboration, and local consistency combination. Finally, in Section 8, we conclude this paper and give some perspectives for further work.

2. An illustration of the language

Suppose that we want to design a solver for Constraint Satisfaction Problems (CSPs) composed of domain constraints (defining the value a variable can assume), and inequations over integer expressions. We are thus concerned with implementation of a solver for problems of the following type:

$$X \leq Y,\ Z \leq 40,\ Y \leq Z,\ Z \in [2..130],\ Y \in [50..100],\ X \in [5..120].$$

Now, suppose that we have heard about a technique of removing impossible values from domains of variables using inequations. This technique is given as two proof rules that reduce the search space without losing any solution:

$$S1 = \frac{X \in [lb_X..rb_X] \ \wedge\ X \leq Y \ \wedge\ Y \in [lb_Y..rb_Y]}{X \in [lb_X..min(rb_X, rb_Y)] \ \wedge\ X \leq Y \ \wedge\ Y \in [lb_Y..rb_Y]}$$

and

$$S2 = \frac{X \in [lb_X..rb_X] \ \wedge\ X \leq Y \ \wedge\ Y \in [lb_Y..rb_Y]}{X \in [lb_X..rb_X] \ \wedge\ X \leq Y \ \wedge\ Y \in [max(lb_X, lb_Y)..rb_Y]}.$$

How to use these rules to design a solver? Such a technique requires Implementation of a mechanism to match the pattern of constraints and some strategy of applications to efficiently apply the rules iteratively until a fixed-point is reached. Our language proposes some help for doing so. A possible solver for this type of constraints using this technique is $dcS1S2$:

$$dcS1S2 \;=\; \mathbf{dc}((S1;S2),\phi_{D\wedge c\wedge Ds})^{\star}.$$

In our language, $\phi_{D\wedge c\wedge Ds}$ is a filter (defined in Section 3) that will select the constraints of the problem that match the head of the rule. $S1$ and $S2$ are solvers. **dc**, ";", and "$\star$" are operators for applying solvers. The expression $dcS1S2$ means:

1. find parts of the constraint that meet our needs, i.e., constraints on which $S1$ and $S2$ can apply. This is performed using the filter $\phi_{D\wedge c\wedge Ds}$,
2. select randomly (selection indicated by **dc**) one of these constraints,
3. apply first $S1$ on the selected constraint, then $S2$ on the result of $S1$ (sequential application of solvers indicated by ";"),
4. iterate Items 1, 2, and 3 until a fixed-point is reached (the "$\star$"), i.e., $S1$ and $S2$ cannot modify the constraint anymore. The result contains the same solutions as the input, but the search space has been reduced.

Now, consider that we have heard about some strategy for finite domains that speeds up reduction of the search space. This strategy is called MinDom: reduction of the smallest domain first can lead to quicker elimination of some branches of the search space. We can easily integrate this strategy using our language:

$$bestS1S2 \;=\; \mathbf{best}((S1;S2),\preceq_{Dom},\phi_{D\wedge c\wedge Ds})^{\star}.$$

This time we use a sorter ($\preceq_{Dom}$ is described in Example 4) that will "order" possible applications of $S1$ and $S2$. The meaning of the expression $bestS1S2$ is:

1. find parts of the constraint that meet our needs ($\phi_{D\wedge c\wedge Ds}$),
2. select the "best" (w.r.t. to the MinDom strategy) possible application. This is performed by $\preceq_{Dom}$ that returns the candidate with the smallest domain,
3. apply $S1$ on the selected constraint, then $S2$,
4. iterate Items 1, 2, and 3 until a fixed-point is reached (the "$\star$").

The result is the same as the previous one, but this time we use a strategy that speeds up resolution.

So far, we can "quickly" reduce the search space. But we need to complete our solver to provide the user with solutions. To this end, we need to consider different branches of the search tree separately. Consider a function *split* that takes as input a domain constraint, and returns a disjunction of domains when possible (this solver is formally defined in Section 5). Roughly, this function is:

$$split(X \in D) = X \in D_1 \vee X \in D_2.$$

We can now create a complete solver (i.e., a solver that not only reduces the search space but also extracts solutions):

$$SolverS1S2 \;=\; (bestS1S2; dc(split, \phi_D))^\star.$$

$SolverS1S2$ first reduces completely the search space using $bestS1S2$. Then, create a disjunction of a domain constraint (the domain constraint is filtered by ϕ_D and splitted by the function *split*). The process reduction-split is iterated on each sub-space until no more split and reduction are possible. The result is a disjunction of possible assignments of variables, i.e., the solutions.

We have informally presented some operators and notions of our language. Some more complex operators (based on parallelism or concurrency) are also provided. The next sections will formally describe the language and some more complex examples.

3. Framework

In this section, we present the basic components of our framework, i.e., *sorters* and *filters*. We first need some definitions fixing our framework.

3.1. Constraints and solvers

Definition 1 (Constraint System). A constraint system is a 4-tuple (Σ, $\mathcal{D}, \mathcal{V}, \mathcal{L}$) where

- Σ is a first-order signature given by a set of function symbols $\mathcal{F}_\Sigma$, and a set of predicate symbols $\mathcal{P}_\Sigma$,
- $\mathcal{D}$ is a Σ-structure (its domain being denoted by $|\mathcal{D}|$),
- $\mathcal{V}$ is an infinite denumerable set of variables, and
- $\mathcal{L}$ is a set of *constraints*: a non-empty set of $(\Sigma, \mathcal{V})$-atomic formulae, called *atomic constraints*, closed under conjunction and disjunction. The *unsatisfiable* constraint is denoted by $\bot$ and the *true* constraint is denoted by $\top$. The set of atomic constraints is denoted by $\mathcal{L}_{At}$.

An assignment is a mapping $\alpha : \mathcal{V} \to |\mathcal{D}|$. The set of all assignments is denoted by $ASS^{\mathcal{V}}_{\mathcal{D}}$. An assignment α extends uniquely to a homomorphism $\underline{\alpha} : T(\Sigma, \mathcal{V}) \to |\mathcal{D}|$. The set of solutions of a constraint $c \in \mathcal{L}$ is the set $Sol_{\mathcal{D}}(c)$ of assignments $\alpha \in ASS^{\mathcal{V}}_{\mathcal{D}}$ such that $\underline{\alpha}(c)$ holds. A constraint c is valid in $\mathcal{D}$ (denoted by $\mathcal{D} \models c$) if $Sol_{\mathcal{D}}(c) = ASS^{\mathcal{V}}_{\mathcal{D}}$. We use $\mathcal{V}ar(c)$ to denote the set of variables from $\mathcal{V}$ occurring in the constraint c. We can now introduce the notion of *a solver*.

Definition 2 (Solver). A *solver* for a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ is a computable function $S : \mathcal{L} \to \mathcal{L}$ such that

1. $\forall C \in \mathcal{L},\ Sol_{\mathcal{D}}(S(C)) \subseteq Sol_{\mathcal{D}}(C)$ (correctness property);
2. $\forall C \in \mathcal{L},\ Sol_{\mathcal{D}}(C) \subseteq Sol_{\mathcal{D}}(S(C))$ (completeness property).

A constraint C is in the solved form with respect to S, if $S(C) = C$.

Given a solver S over a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, we extend S to a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L}')$, where $\mathcal{L} \subseteq \mathcal{L}'$, in the following way: $\forall\ C \in \mathcal{L}' \setminus \mathcal{L}$, $S(C) = C$. We say that a constraint C is in the solved form with respect to S, if $S(C) = C$.

Example 1. Consider the constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ such that the constraint symbols (i.e., the predicate symbols) c of arity n and $\in$ are in Σ, $|\mathcal{D}|$ is finite. Constraints of the form $X \in D$ are called domain constraints, and they are widely used in CSPs: the set D (called the *domain* of X) specifies the values of $|\mathcal{D}|$ the variable X can take.

Consider now the *LocalConsistency* function that takes as input the following pattern of constraints

$$\begin{aligned} x_i \in D_i \wedge c(x_1, \ldots, x_i, \ldots, x_n)\ &\wedge x_1 \in D_1\ \wedge \ldots \wedge\ x_{i-1} \in D_{i-1} \\ &\wedge x_{i+1} \in D_{i+1}\ \wedge \ldots \wedge\ x_n \in D_n \end{aligned}$$

and returns

$$\begin{aligned} x_i \in D_i' \wedge c(x_1, \ldots, x_i, \ldots, x_n)\ &\wedge x_1 \in D_1\ \wedge \ldots \wedge\ x_{i-1} \in D_{i-1} \\ &\wedge x_{i+1} \in D_{i+1}\ \wedge \ldots \wedge\ x_n \in D_n, \end{aligned}$$

where

$$\begin{aligned} D_i' = \{v_i \in D_i \mid\ &(\exists\, v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1}, \\ &\exists v_{i+1} \in D_{i+1}, \ldots, \exists v_n \in D_n) :\ c(v_1, \ldots, v_i, \ldots, v_n)\}. \end{aligned}$$

Then, *LocalConsistency* is a solver, i.e., it removes impossible values from the domain of x_i using the constraint c, but preserves solutions of c. This solver can be efficiently implemented for several standard constraints, such as $=$ and $\leq$ over finite domains (i.e., generally, integers that can be represented in a computer). *LocalConsistency* is used in Section 5.1.

3.2. Syntactical forms and sub-constraints

On the previous example, we have seen that a solver cannot always be applied on a "complete" constraint but only on a part of it ($S_{\leq}$ could be applied only on a special pattern of constraints). Thus, to define specific parts of a constraint, we introduce the notions of *a syntactical form* and *a sub-constraint.*

Definition 3 (Syntactical Forms and Sub-constraints). We say that C' is a *syntactical form* of C, denoted by $C' \approx C$, if $C' = C$ modulo the associativity and commutativity of $\wedge$ and $\vee$, and the distributivity of $\wedge$ on $\vee$ and of $\vee$ on $\wedge$ [1]. We say that $C' \in \mathcal{L}$ is a *sub-constraint* of C, denoted by $C_{[C']}$, if

- $C = C'$,
- or $\exists C_1 \in \mathcal{L}$, $\omega \in \{\wedge, \vee\}$, $C = C_1 \omega C'$,
- or $\exists C_1 \in \mathcal{L}$, $\omega \in \{\wedge, \vee\}$, $C = C' \omega C_1$,
- or $\exists C_1, C_2 \in \mathcal{L}$, $\omega \in \{\wedge, \vee\}$, $C = C_1 \omega C_2$ and ($C_{1[C']}$ or $C_{2[C']}$).

A couple (C'', C') such that C'' is a sub-constraint of C' and $C' \approx C$ is called an *applicant* of C. We denote by $\mathcal{SF}(C)$ the finite set of all the syntactical forms of a constraint C: $\mathcal{SF}(C) = \{C' |\, C' \approx C\}$[2]. We denote by $\mathcal{LA}$ the set of all the lists of applicants, and by $\mathcal{LC}$ the set of all the lists of constraints. Generally, we will use LA (respectively LC) to denote a list of applicants (respectively constraints). We denote by $\mathcal{P}(\mathcal{L} \times \mathcal{L})$ the power-set of all the sets of couples of constraints. $\mathcal{A}tom(C)$ denotes the set of atomic constraints that occur in C: $\{c | c \in \mathcal{L}_{At}$ and $C_{[c]}\}$.

3.3. Filters and sorters

We now define the basic components of our strategy language: filters to select specific parts of a constraint, and sorters to classify the elements of a list w.r.t. a given ordering. These transformations are generally hidden in the implementation of solvers.

We introduce the notion of a filter for two main reasons. A solver can, in general, be applied on several parts of a constraint [11]. Second, when dealing with solver collaborations, in general, a single solver is not able to treat the complete constraint [21]. In both cases, we want to identify the sub-parts of the constraint that the solver is actually able to handle. The usefulness of filters is clear when, for example, we want to manipulate only the domain constraints like $X \in D_X$ from a set of constraints C in order to carry out enumeration. Also, when one is interested in verifying the local consistency (such as in the solver of Example 1), it is necessary to select sub-constraints. In this case, a sub-constraint is the conjunction of a domain constraint, an atomic constraint, and a conjunction of domain constraints, i.e., an atomic constraint, and all the domain constraints of the variables occurring in it (see filter of Example 2).

[1] We consider that "=" is purely syntactic.

[2] The ACD theory defines a finite set of quotient classes that we can effectively filter.

Once we have identified different parts of the constraint on which a given solver can be applied, we generally want to select some of them based on a given criterion, i.e., the best of these parts in order to "optimize" the application of the solver. Thus, we introduce the notion of a sorter associated with the concept of a strategy.

For example, when solving constraints, we sometimes are interested in choosing a variable that can take the minimum or the maximum number of values. If we suppose that we can already select all the domain constraints like $X \in D_X$ from a set of constraints C using the notion of a filter, we can easily imagine a sorter to implement the minimum or the maximum domain criterion (see the sorter of Example 4).

Definition 4 (Filter). Let $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ be a constraint system. Then, a filter ϕ on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ is a computable function $\phi : \mathcal{L} \to \mathcal{P}(\mathcal{L} \times \mathcal{L})$ such that

$$\forall C \in \mathcal{L},\ \phi(C) = \{(Cf_i, C_i), \ldots, (Cf_n, C_n)\},$$

where $\forall i \in [1, n]$, $C \approx C_i$ (C_i is a syntactical form of C), and $C_{i[Cf_i]}$ (Cf_i is a sub-constraint of C_i).

The elements of $\phi(C)$ are called *candidates*. We define the filter Id which returns the initial set of constraints. Given the filters ϕ and ϕ' on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, we say that

- ϕ is *selective* if $\forall C \in \mathcal{L}$, $\phi(C) = \{(Cf_1, C_1), \ldots, (Cf_n, C_n)\}$ such that $\forall i, j \in [1, \ldots, n] \times [1, \ldots, n], i \neq j$, $\mathcal{A}tom(Cf_i) \cap \mathcal{A}tom(Cf_j) = \emptyset$;
- ϕ is *stable* if $\forall C \in \mathcal{L}$, $\phi(C) = \{(Cf_1, C'), \ldots, (Cf_n, C')\}$;
- ϕ and ϕ' are *disjoint* if $\forall C \in \mathcal{L}$, $\phi(C) = \{(Cf_1, C_1), \ldots, (Cf_n, C_n)\}$, and $\phi'(C) = \{(Cf'_1, C'_1), \ldots, (Cf'_m, C'_m)\}$, s.t. $\forall (i, j) \in [1, \ldots, n] \times [1, \ldots, m]$, $\mathcal{A}tom(Cf_i) \cap \mathcal{A}tom(Cf'_j) = \emptyset$.

Property 1. Let ϕ_1 and ϕ_2 be two filters on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$. Then, $\phi_1; \phi_2$ defined by

$$\forall C \in \mathcal{L}, \phi_1; \phi_2(C) = \phi_1(C) \cap \phi_2(C)$$

is a filter on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$.

Property 1 enables one to design complex filters using more basic filters.

Example 2. We define a simple filter for the domain constraints

$$\forall C \in \mathcal{L},\ \phi_D(C) = \{(c, C) | C_{[c]} \text{ and } \exists X \in \mathcal{V},\ c = (X \in D_X)\}.$$

The filter ϕ_D is stable and selective. We denote by $\mathcal{L}_{Dom}$ the elements of $\mathcal{L}_{At}$ resulting from the application of this filter. We will use this notation in other examples.

Example 3. We now consider patterns of constraints (such as in the solver of Example 1). The utility of this filter will be clarified in Section 5. We want to filter sub-constraints that are the conjunction of a domain constraint, an atomic constraint, and a conjunction of domain constraints, i.e., an atomic constraint, and all the domain constraints of the variables occurring in it. $\forall C \in \mathcal{L}$, $\phi_{D\wedge c\wedge Ds}(C) \subseteq \mathcal{L}^2$ and $\phi_{D\wedge c\wedge Ds}(C)$ is defined as follows:

1. Patterns:
$$\begin{array}{rl}(C'', C') \in \phi_{D\wedge c\wedge Ds}(C) \Rightarrow & C'' = (X \in D_X) \\ & \wedge\, c \ \bigwedge_{Y\in \mathcal{V}ar(c)\setminus\{X\}} Y \in D_Y \\ & \wedge\, c \in \mathcal{L}_{At} \setminus \mathcal{L}_{Dom} \\ & \wedge\, C' \in SF(C) \\ & \wedge\, C'_{[C'']} \\ & \wedge\, X \in \mathcal{V}ar(c).\end{array}$$
2. Context-free:
$$((C', C_1) \in \phi_{D\wedge c\wedge Ds}(C) \wedge (C', C_2) \in \phi_{D\wedge c\wedge Ds}(C)) \Rightarrow C_1 = C_2.$$
3. Commutative-free:
$$\left.\begin{array}{l}(X \in D_X \wedge c \wedge C''_1, C_1) \in \phi_{D\wedge c\wedge Ds}(C) \\ \wedge\, (X \in D_X \wedge c \wedge C''_2, C_2) \in \phi_{D\wedge c\wedge Ds}(C)\end{array}\right\} \Rightarrow C''_1 \approx C''_2.$$

Item 1 requires that elements of $\phi_{D\wedge c\wedge Ds}(C)$ have some syntactical properties, i.e., form a pattern of constraints; in Item 2, we do not want to consider several times the same sub-constraints issued from different syntactical forms of C; and finally, in Item 3, we specify that the ordering of the conjunction of domain constraints is not relevant.

Item 2 and 3 are not mandatory, but they reduce the number of applicants. This definition does not provide uniqueness of the filter. Depending on our needs, we can consider (1) adding the requirements to define one set of applicants per constraint, (2) removing Item 2 and 3, or (3) selecting one of the sets corresponding to the definition.

For example, consider the problem of solving CSPs and a function S (or a transformation rule) which reduces the domain of one variable using one constraint. Then, for each constraint of the CSP and each variable of this constraint, we can consider a possible application of S.

Definition 5 (Sorter). A *sorter Sorter*, w.r.t. a partial ordering $\preceq$, for a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ is a computable function $Sorter : \preceq \times \mathcal{P}(\mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{LA}$, such that $\forall\{(Cf_{i_1}, C_{i_1}), \ldots, (Cf_{i_n}, C_{i_n})\} \in \mathcal{P}(\mathcal{L} \times \mathcal{L})$

1. $Sorter(\preceq, \{(Cf_{i_1}, C_{i_1}), \ldots, (Cf_{i_n}, C_{i_n})\}) = [(Cf_1, C_1), \ldots, (Cf_n, C_n)]$;
2. $\forall k \in [1, \ldots, n], \exists j \in [1, \ldots, n]$, $Cf_{i_j} = Cf_k$ and $C_{i_j} = C_k$;

3. $\forall j \in [1, \ldots, n-1], Cf_j \preceq Cf_{j+1}$.

Remark 1. We assume that a sorter is deterministic, i.e., if L is a set of applicants, each application of Sorter on L will always return the same list of applicants.

Example 4 (MaxDom and MinDom sorters). The $\preceq_{Dom}$ ordering is based on the width of the domain constraint[3]. For atomic domain constraints, $\preceq_{Dom}$ is straight-forward, but we may need to consider this ordering for more complex constraints (e.g., patterns of constraints issued from filters). We define the function ω, the width of a constraint, as follows:

- if $c \in \mathcal{L}_{Dom}$ and $c = X \in D$ then $\omega(c) = width(D)$,
- if $c \in \mathcal{L}_{At} \setminus \mathcal{L}_{Dom}$ then $\omega(c) = -1$,
- if $C = c \wedge C'$ or $C = c \vee C'$ then $\omega(C) = \omega(c)$.

$\preceq_{Dom}$ is now defined by

$$\forall C, C' \in \mathcal{L},\ C \preceq_{Dom} C' \ \text{ if } \ \omega(C) \leq \omega(C').$$

The sorter MinDom (respectively, MaxDom) is defined by the $\preceq_{Dom}$ ordering (respectively, $\succeq_{Dom}$, the reverse ordering of $\preceq_{Dom}$).

4. The language

In this section, we define the operators of our strategy language. They are used to apply solvers to selected parts of constraints. Most of the operators are based on the same mechanism when applied to a constraint C:

1. A set SC of candidates is built using the filter ϕ on C.
2. The set SC is sorted using the partial order $\preceq$. We obtain LC, a sorted list of candidates.
3. The solver S is applied to one (e.g., the "best" w.r.t. $\preceq$) or several elements of LC.
4. Each occurrence of the sub-constraint(s) modified by S is replaced (substituted) in its corresponding (w.r.t. candidates) syntactical form of C.

In the following, we consider a given constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, solvers $S_1, \ldots, S_n$, filters $\phi_1, \ldots, \phi_n$, and partial orders $\preceq_1, \ldots, \preceq_n$. We denote by $C\{c' \mapsto c''\}$ the substitution of the sub-constraint c' of C by c''. Note that a substitution applies to every occurrence of a sub-constraint.

[3]For interval domains, $width(D)$ can be the difference between the upper and the lower bound. On the other hand, for domains that are sets of elements, the width can be defined as the cardinality of the set. In every case, width is a numeric value.

4.1. Basic operators

These operators are analogous to function compositions and allow us to design solvers by combining the "basic" functions (non-decomposable solvers), or to create solver collaborations by combining component solvers. Consider two solvers S_i and S_j. Then, for all $C \in \mathcal{L}$

- $S_i^0(C) = C$ (*Identity*),
- $S_i; S_j(C) = S_j(S_i(C))$ (*solver concatenation*),
- $S_i^n(C) = S_i^{n-1}; S_i(C)$ if $n > 0$ (*solver iteration*),
- $S_i^\star(C) = S_i^n(C)$ such that $S_i^{n+1}(C) = S_i^n(C)$ (*solver fixed-point*),
- $(S_i, S_j)(C) = S_i(C)$ or $S_j(C)$ (*solver don't-care*).

Property 2. Let S_i and S_j be two solvers. Then, $S_i; S_j$, S_i^n, $S_i^\star$, and (S_i, S_j) are solvers.

4.2. Best and random applications of solvers

The following two operators apply solvers to specific components of constraints.

Don't care application of a solver: the **dc** operator restricts the use of the solver S_i to one randomly chosen sub-constraint of a syntactical form of C (obtained using the filter ϕ).

$$\forall C \in \mathcal{L}, \mathbf{dc}(S_i, \phi)(C) = C',$$

where

- $[(Cf_1, C_1), \ldots, (Cf_n, C_n)] = \phi(C)$;
- if there exists $i \in [1, \ldots, n]$ such that $S_i(Cf_i) \neq Cf_i$, then $C' = C_i\{Cf_i \mapsto S_i(Cf_i)\}$, otherwise $C' = C$.

Best application of a solver: the **best** operator restricts the use of the solver S_i to the best (w.r.t. the partial order $\preceq$) sub-constraint of a syntactical form of C (obtained using the filter ϕ) that S_i is able to modify.

$$\forall C \in \mathcal{L}, \mathbf{best}(S_i, \preceq, \phi)(C) = C',$$

where

- $[(Cf_1, C_1), \ldots, (Cf_n, C_n)] = \mathit{Sorter}(\preceq, \phi(C))$;
- if there exists $i \in [1, \ldots, n]$, such that $S_i(Cf_i) \neq Cf_i$, and $\forall j \in [1, \ldots, n]$ $(S_i(Cf_j) \neq Cf_j \Rightarrow i \leq j)$ then $C' = C_i\{Cf_i \mapsto S_i(Cf_i)\}$, otherwise $C' = C$.

4.3. Concurrent and parallel applications of solvers

These two operators manage several solvers. The first one selects the result of one solver depending on a required constraint property, the second one composes the resulting constraints based on the results of each solver. A constraint property p on a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ is a function from constraints to Booleans (i.e., $p : \mathcal{L} \to \mathcal{B}oolean$).

Concurrent application of solvers: the **pcc** operator applies one of the solvers S_i once and returns a constraint that verifies the property p.

$$\forall C \in \mathcal{L}, \mathbf{pcc}(p, [S_1, \preceq_1, \phi_1], \ldots, [S_n, \preceq_n, \phi_n])(C) = C',$$

where

- for all $i \in [1, \ldots, n]$ $[(Cf_{i,1}, C_{i,1}), \ldots, (Cf_{i,m_i}, C_{i,m_i})] = Sorter(\preceq_i, \phi_i(C))$;
- if there exists $(i,j) \in [1, \ldots, n] \times [1, \ldots, m_i]$ such that $p(S_i(Cf_{i,j}))$, and $S_i(Cf_{i,j}) \neq Cf_{i,j}$, then $C' = C_{i,j}\{Cf_{i,j} \mapsto S_i(Cf_{i,j})\}$, otherwise $C' = C$.

Parallel applications of solvers: we assume the filters $\phi_1, \ldots, \phi_n$ to be stable and pairwise disjoint. The **bp** operator applies n solvers $S_1, \ldots, S_n$ on n sub-constraints of one syntactical form of a constraint.

$$\forall C \in \mathcal{L}, \mathbf{bp}([S_1, \preceq_1, \phi_1], \ldots, [S_n, \preceq_n, \phi_n])(C) = C',$$

where

- for all $i \in [1, \ldots, n]$ $[(Cf_{i,1}, C''), \ldots, (Cf_{i,m_i}, C'')] = Sorter(\preceq_i, \phi_i(C))$;
- for all $i \in [1, \ldots, n]$, if there exists $j \in [1, \ldots, m_i]$, s.t. $S_i(Cf_{i_j}) \neq Cf_{i_j}$, and for all $k < j$, $S_i(Cf_{i_k}) = Cf_{i_k}$, then $\sigma_i = \{Cf_{i,i_j} \mapsto S_i(Cf_{i,i_j})\}$, else $\sigma_i = \emptyset$;
- $C' = C''\sigma$, where $\sigma = \bigcup_{i \in [1, \ldots, n]} \sigma_i$.

4.4. Associating sub-problems with distinct solvers

Finally, we present two operators to apply a solver on each component of a conjunction or disjunction of constraints. The result is obtained by conjunction or disjunction of the resulting constraints, respectively. These operators enable parallel computation, and standard OR_parallel computation.

To this end, the notion of *a separator* is introduced. It can be seen as a pre-processing for parallel computation. Separators are mainly defined to manipulate the elements of conjunctions and disjunctions of constraints as elements of lists[4]. Each element of the list will then be treated separately but in parallel before gathering (conjunction or disjunction) all the results.

Definition 6 (Separators). A $\wedge$***_separator*** δ is a function $\delta : \mathcal{L} \to \mathcal{LC}$ such that

$$\forall\ C \in \mathcal{L},\ \exists n \in \mathbb{N},\ \delta(C) = [C_1, \ldots, C_n] \text{ where } C \approx C_1 \wedge \ldots \wedge C_n.$$

Similarly, a $\vee$***_separator*** δ is a function $\delta : \mathcal{L} \to \mathcal{LC}$ such that

$$\forall\ C \in \mathcal{L},\ \exists n \in \mathbb{N},\ \delta(C) = [C_1, \ldots, C_n] \text{ where } C \approx C_1 \vee \ldots \vee C_n.$$

Example 5. Consider a disjunctive problem in which variables can assume several domains. This is a standard case when representing at once the exploration of several branches of a search space. Then, we would like to be able to consider every branch of the search space (see Section 5 for illustration). Thus, we consider a $\vee$**_separator** named $CSP_\vee$, defined by

$$\forall C \in \mathcal{L},\ CSP_\vee(C) = [C_1, \ldots, C_n],$$

$$\text{such that } C \approx C_1 \vee \ldots \vee C_n \text{ and } \begin{cases} C_1 & = & X \in D_X^1 \wedge C' \\ \vdots & \vdots & \vdots \\ C_n & = & X \in D_X^n \wedge C' \end{cases}.$$

Conjunctive sub-problems: the $\wedge$**_p** operator applies (in parallel) the solver S_i to several conjuncts (determined by $\delta_\wedge$) of the constraint C and the final result is obtained by conjunction of the results computed in parallel:

$$\forall C \in \mathcal{L}, \wedge\_\mathbf{p}(S_i, \delta_\wedge)(C) = C',$$

where

- $[C_1, \ldots, C_n] = \delta_\wedge(C)$,
- $C' = S_i(C_1) \wedge \ldots \wedge S_i(C_n)$.

Disjunctive sub-problems: the $\vee$**_p** operator is analogous to $\wedge$**_p**, but $\delta_\vee$ determines disjuncts, and the final result is the disjunction of the results computed in parallel:

$$\forall C \in \mathcal{L}, \vee\_\mathbf{p}(S_i, \delta_\vee)(C) = C',$$

where

[4]Lists enable us to sort and explore the search tree in a deterministic way. This is particularly important when we consider sequential implementations, i.e., the branches are processed sequentially. In such cases, the use of sets leads to non-deterministic search.

- $[C_1, \ldots, C_n] = \delta_\vee(C)$,
- $C' = S_i(C_1) \vee \ldots \vee S_i(C_n)$.

In spite of its simplicity, the following property is essential, since it allows us to manipulate the basic functions and solvers at the same level, and thus to create solvers and solver collaborations with the same strategy language.

Property 3. Consider n solvers $S_1, \ldots, S_n$, n filters $\phi_1, \ldots, \phi_n$, n partial orders $\preceq_1, \ldots, \preceq_n$, a constraint property p, and separators $\delta_\wedge$ and $\delta_\vee$. Then, $\mathbf{dc}(S_i, \phi)$, $\mathbf{best}(S_i, \preceq, \phi)$, $\mathbf{pcc}(p, (S_1, \preceq_1, \phi_1), \ldots, (S_n, \preceq_n, \phi_n))$, $\mathbf{bp}((S_1, \preceq_1, \phi_1), \ldots, (S_n, \preceq_n, \phi_n))$ (assuming $\phi_1, \ldots, \phi_n$ to be stable and pairwise disjoint), $\wedge\_\mathbf{p}(S_i, \delta_\wedge)$, and $\vee\_\mathbf{p}(S_i, \delta_\vee)$ are solvers.

5. A generic propagation-based solver

A CSP is given by a set of constraints together with a set of domain constraints, one for each variable of the problem. Constraint propagation is a widely recognized concept aimed to reduce a CSP into an equivalent but simpler one (meaning the search space is reduced, but no solution is lost) by narrowing the domains of variables until a fixed-point is reached. However, constraint propagation must be interleaved with a splitting mechanism in order to compose a complete solver, i.e., a solver able not only to reduce the problem, but also to extract solutions. This mechanism works by splitting the domain of a variable into (sub)domains.

The reduction process is performed by *domain reduction functions* in the scheme of K. R. Apt [1], and by *narrowing operators* in the framework of F. Benhamou [5]. These reduction functions or narrowing operators are managed by a propagator (such as a set for a don't care application, or a list, e.g., for a MinDom strategy) that composes the reduction strategy. We now present an implementation of these two frameworks using our strategy language. Then we instantiate this generic solver to solve CSPs over finite domains and interval real numbers.

Reduction: We first consider *g_narrow*, a generic reduction solver that takes as input a domain constraint (the domain of the variable to be reduced), a constraint (the information used to reduce the variable), and the domain constraint of the variables occuring in the constraint (this information is required for most, if not for all, narrowing operators). This solver requires the $\phi_{D\wedge c\wedge Ds}$ filter of Example 3. The *dc_g_narrow* is the don't-care complete reduction of a CSP:

$$dc\_g\_narrow \;=\; \mathbf{dc}(g\_narrow, \phi_{D\wedge c\wedge Ds})^\star.$$

The $MaxD\_g\_narrow$ uses the $\succeq_{Dom}$ sorter defined in Example 4 and implements the MaxDom reduction strategy:

$$MaxD\_g\_narrow \;=\; \mathbf{best}(g\_narrow, \succeq_{Dom}, \phi_{D \wedge c \wedge Ds})^{\star}.$$

Splitting mechanism: We consider the g_split generic solver which transforms a domain constraint into a disjunction of two domain constraints if the width of the domain is greater than or equal to a "minimal" width ϵ. For continuous domains, ϵ generally represents the smallest difference that can be computed between two numbers. For discrete domains, ϵ is generally set to 1. The solver $g\_split : \mathcal{L} \to \mathcal{L}$ is defined as follows, using the function ω that gives the width of a constraint (see Example 4). For all $c = X \in D$ from $\mathcal{L}$,

- if $c \in \mathcal{L}_{Dom}$ such that $width(c) \geq \epsilon$, then

$$g\_split(c) = X \in D' \vee X \in D'', \quad \text{where} \quad D = D' \cup D''\,^{5},$$

- otherwise, $g\_split(c) = c$.

The following solver splits a randomly chosen domain,

$$dc\_g\_split \;=\; \mathbf{dc}(g\_split, \phi_D),$$

whereas $MaxD\_g\_split$ splits the biggest current domain of the CSP:

$$MaxD\_g\_split \;=\; \mathbf{best}(g\_split, \succeq_{Dom}, \phi_D).$$

Note that in both split solvers, we use the ϕ_D filter defined in Example 2.

Generic propagation-based solvers: Here we give some generic solvers implementing the standard strategies. Note that, using other operators, filters, and sorters, we can easily design other standard and non-standard strategies. The first solver

$$dc\_g\_prop \;=\; (dc\_g\_narrow\,;\, dc\_g\_split)^{\star}$$

represents a basic strategy in which no specific selection (for reduction and splitting) is performed. On the other hand,

$$MaxD\_g\_prop \;=\; (MaxD\_g\_narrow\,;\, MaxD\_g\_split)^{\star}$$

is a complete propagation-based solver implementing a MaxDom strategy. Note that we similarly obtain a $MinD\_g\_prop$ solver by replacing the $\succeq_{Dom}$

[5] Generally we also enforce it with $D' \cap D'' = \emptyset$.

sorter by the $\preceq_{Dom}$ sorter. The solving process is neither depth-first, nor breadth-first, but MaxDom first, i.e., we reduce one branch, and then we eventually choose another branch (the one with the biggest domain) to explore.

We are now concerned with a homogeneous exploration of branches. We consider the $\vee$**_separator** $CSP_\vee$ defined in Example 5. We now get another generic solver:

$$MaxD\_\vee\_g\_prop \;=\; \vee\_\mathbf{p}(MaxD\_g\_narrow\,;\,MaxD\_g\_split, CSP_\vee)^\star.$$

Depending on the implementation of the $\vee$**_p** operator, we will obtain a depth-first search (sequential implementation) or a parallel exploration of every branches (parallel implementation).

Using δ_{Var}, a $\wedge$**_separator** which splits a set of constraints into n variable-disjoint subsets of constraints, the application of $MaxD\_g\_prop$ can be improved when solving CSPs that can be decomposed:

$$Sp\_MaxD\_g\_prop \;=\; \wedge\_\mathbf{p}(MaxD\_g\_prop, \delta_{Var}).$$

In this way, we are solving several CSPs in parallel. An obvious advantage is to deal with simpler problems. The solution to the original problem will be in the union of the solutions to all subproblems.

In the next sub-sections, we instantiate the generic solvers g_narrow and g_split in order to obtain solvers over finite domains and interval real numbers.

5.1. Solving constraints over finite domains

A CSP P over finite domains is any conjunction of formulae of the form:

$$\bigwedge_{x_i \in \mathcal{X}} (x_i \in D_{xi}) \wedge C,$$

where a domain constraint $x_i \in D_{xi}$ is created for each variable x_i occurring in the constraint C, D_{xi} being a finite set of values.

First, we just have to instantiate the g_narrow generic solver with the solver $LocalConsistency$ described in Example 1. dc_g_narrow and $MaxD\_g\_narrow$ becomes two solvers that enforce arc-consistency [18].

Second, we instantiate g_split with $\epsilon = 1$, and $width(D) = card(D)$, when D is a domain, and we enforce that $D' \cap D'' = \emptyset$.

With these instantiations, dc_g_prop becomes a finite domain constraint solver that implements the standard full lookahead strategy [17]. Now, if we consider $MinD\_g\_prop$ instead of dc_g_prop, then we obtain a full lookahead strategy combined with a MinDom strategy (i.e., a standard strategy for finite domains aimed to find quickly inconsistencies in the set of constraints).

However, we can consider some more specific finite domain strategies, like the forward checking [17]. This heuristic, when enforcing local consistency, takes into account just the constraints that are directly related to the splitted variable. We consider another filter $\phi_{D \wedge c \wedge C \wedge Ds}$: this filter returns a domain constraint D over a variable X, a constraint c that contains X, all the constraints (the conjunction C) that contain X (except c), and all the domain constraints of the variables that appear in $c \wedge C$. We also consider an extension g_split' of the instantiation of the solver g_split that is applied on the result of the filter $\phi_{D \wedge c \wedge C \wedge Ds}$. When applied to a constraint $D \wedge c \wedge C \wedge Ds$, g_split' returns $g\_split(D) \wedge c \wedge C \wedge Ds$. We can formulate Forward Checking using dc_g_narrow instantiated with $LocalConsistency$ as follows:

$$\begin{aligned} &ForwardChecking = \\ &\quad dc\_g\_narrow \ ; \ \mathbf{dc}((g\_split'; dc\_g\_narrow), \phi_{D \wedge c \wedge C \wedge Ds})^*. \end{aligned}$$

We can obviously consider full lookahead and forward checking strategies using a MinDom strategy: to this end, we just have to consider $MinD\_g\_narrow$ and $MinD\_g\_split$ instead of dc_g_narrow and dc_g_split respectively. We can also consider $Sp\_MaxD\_g\_prop$ to separate the problem into sub-problems and to create numerous new strategies using the same solvers but different strategy operators of our language.

5.2. Solving constraints over real numbers

We now design solvers for non-linear constraints over real interval arithmetic. In the following, a CSP P is any conjunction of formulae of the form

$$\bigwedge_{x_i \in \mathcal{X}} (x_i \in D_{xi}) \wedge C,$$

where a domain constraint $x_i \in D_{xi}$ is created for each variable x_i occurring in the set of constraints C, D_{xi} being an interval of real numbers. Constraints are equalities, inequalities, and inequations of non-linear terms built over intervals of real numbers and the function symbols $+, -, *, /, \hat{}$, sin, and cos.

Consider the function b_c which, given a non-linear constraint $c \in \mathcal{L}_{At} \setminus \mathcal{L}_{Dom}$, the domain D of a variable $X \in \mathcal{V}ar(c)$, and the domains of the other variables of $\mathcal{V}ar(c)$, returns a smaller domain for X such that c is box-consistent [28] with respect to X [6].

[6]Computing b_c generally consists in applying the interval Newton method combined with a "local" splitting mechanism to push the left and right bounds of the interval.

We now define the solver $drf : \mathcal{L} \to \mathcal{L}$. For all $C \in \mathcal{L}$, we compute $drf(C)$ depending on the syntactical form of C:

- if $C = X \in D_X \wedge c \wedge \bigwedge_{Y \in \mathcal{V}ar(c) \setminus \{X\}} Y \in D_Y$, where $c \in \mathcal{L}_{At} \setminus \mathcal{L}_{Dom}$, then
$$drf(C) = X \in D'_X \wedge c \wedge \bigwedge_{Y \in \mathcal{V}ar(c) \setminus \{X\}} Y \in D_Y,$$
where $D'_X = b\_c(c, D_X, \{D_Y | Y \in \mathcal{V}ar(c) \setminus \{X\}\})$,
- otherwise, $drf(C) = C$.

We instantiate the solver *g_narrow* by the solver *drf*. *dc_g_narrow* becomes a solver that enforces box-consistency of a set of non-linear constraints, i.e., each constraint is box-consitent with respect to each of its variables. *MaxD_g_narrow* enforces box-consistency using a MaxDom strategy (i.e., a standard strategy for numeric real number solver).

In order to isolate solutions, we need to instantiate *g_split*. We take $\epsilon = 10^{-8}$, the precision of computation of solutions. The width function is instantiated by: for all intervals $I = [a, b]$, $\omega(I) = b - a$. Finally, we enforce that $D' \cap D'' = \emptyset$. Thus, *dc_g_prop* becomes a solver that returns solutions with a precision of 8 decimals.

MaxD_$\vee$_g_prop becomes a similar solver that separately explores every branch. On the other hand, *Sp_MaxD_g_prop* creates disjoint sub-problems before any reduction.

6. Optimization problems over finite domains

We now concentrate on an extension of a CSP called Constraint Satisfaction Optimization Problem (CSOP). CSOP consists in finding an optimal (i.e., maximal or minimal) value for a given function, such that a set of constraints is satisfied [27]. The work of Bockmayr and Kasper [7] explains the approach generally used by the constraint solving community to deal with this problem. In this section, we first explain two approaches for solving CSOPs, and then we show how they can be combined using our strategy language.

A CSOP can be described by a tuple $\langle P, f, lb, ub \rangle$ representing a CSP, an optimization function, and the lower and upper bounds of this function. Without loss of generality, we consider the case of minimization of a function f over integers. To deal with this problem, we consider two approaches, both of them requiring an initial step verifying that $Sol(C \wedge f \leq ub) \neq \emptyset$, i.e., there exists a solution to the constraint C satisfying the additional constraint $f \leq ub$.

The first approach consists in applying the following rule until it cannot be applied any more:

$$\langle P, f, lb, ub\rangle \quad \to \quad \langle P, f, lb, \alpha(f)\rangle \textbf{ if } \alpha \in Sol(C \wedge f < ub).$$

Each iteration of this rule tries to decrease the upper bound ub by at least one unit until an unsatisfiable problem is obtained. That is why we call this technique *satisfiability to unsatisfiability*. The minimum value of the function f represents the upper bound of the last successful application of this rule. Thus, we define the solver *MinSatToUnsat* implementing this approach. We do not detail here this definition, but it is obvious that for solving CSPs, as needed by this approach, we can use the solvers defined in Section 5.1.

The second approach applies the following rules until they cannot be applied any more:

$$\begin{array}{ll} \langle P, f, lb, ub\rangle \to \langle P, f, lb, \alpha(f)\rangle \textbf{ if } \alpha \in Sol(C \wedge f < \frac{(lb+ub)}{2}), & \\ \langle P, f, lb, ub\rangle \to \langle P, f, \frac{(lb+ub)}{2}, ub\rangle \textbf{ if } \ lb \neq ub & \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and } \ Sol(C \wedge f < \frac{(lb+ub)}{2}) = \emptyset. & \end{array}$$

The first rule tries to find a new value for the upper bound ub and reduces, at least in half, the range of possible values of the function f each time a new solution is obtained[7]. The second rule similarly updates the lower bound lb in the opposite situation. We call this approach *binary splitting* and define the solver *MinSplitting* implementing it.

Concerning the behavior of these strategies, we can note that the strategy *MinSatToUnsat* is very slow for reaching the minimal value of f, when it is located far from the initial upper bound. On the other hand, applying the strategy *MinSplitting*, the same situation happens when the minimal value of f is close to the initial upper bound. Since it is not evident where the optimal solution is located, an *a priori* choice between these approaches is generally impossible. To improve the performance of these two basic solvers, we can make them collaborate in order to profit from the advantages of both of them, and to avoid their drawbacks.

A first scheme of cooperation is expressed by the strategy *SeqOpt*:

$$SeqOpt = (MinSatToUnsat; MinSplitting)^{\star}.$$

With the strategy *SeqOpt*, both solvers are executed sequentially. Its obvious disadvantage is that it leaves a solver inactive, while the other one is working. Moreover, due to the exponential complexity of the problem under

[7]Of course, we can think of different ratios, thus, the first approach can be seen as a particular case of the second one.

consideration, the whole process could be blocked if one solver cannot find a solution. To avoid this situation, we can run them concurrently, updating the current solution as soon as a new one is available, and stopping the other solver.

$$\begin{aligned} ParOpt = \mathbf{pcc}(first,\ & [MinSatToUnsat, None, Id], \\ & [MinSplitting, None, Id])^{\star}. \end{aligned}$$

We do not filter the initial set of constraints and so we do not have any sorter. In this case, we are interested in the solver that will be faster, that is why we use the *first* property [8]. With this strategy, a solver never waits for a solution coming from the other one. In the worst case (i.e., all solutions are read from the same elementary solver until the final solution is obtained), the performance of the $ParOpt$ solver is the same as if one of the elementary solvers ran independently.

7. Solver collaborations

7.1. Combining symbolic and numerical methods

Here we consider the systems of non-linear constraints and two solvers. Gröbner bases computation [8] (i.e., the *gb* solver) transforms a set of multivariate polynomial equalities into a normal form from which solutions can be derived easier than from the initial set. The second solver, *int*, is a propagation-based numerical solver over the real numbers (e.g., one of the solvers presented in Section 5.2). We assume that every constraint of the CSPs we consider can be processed by *int*.

It is generally very efficient to pre-process a CSP with symbolic rewriting techniques before applying a propagation-based solver. In fact, the preprocessing may add redundant constraints (in order to speed-up propagation), simplify constraints, deduce some univariate constraints (whose solutions can easily be extracted by propagation), and reduce the variable dependency problem.

Thus, we consider *sc*, a simple collaboration where Gröbner bases computation pre-processes the equality constraints before the interval solver is applied on the whole CSP:

$$sc = \mathbf{dc}(gb, \phi_{=}); int,$$

where the filter $\phi_{=}$ selects equalities of polynomials.

Consider, for example, the following problem:

[8]Here, since we consider parallel computation, we extend the properties of constraints to the properties of constraints and computations.

$$x^3 - x * y^2 + 2 = 0 \ \wedge \ x^2 - y^2 + 2 = 0 \ \wedge \ y > 0.$$

Most of the solvers based on propagation require splitting to isolate the solutions of this CSP. However, using gb (with a lexicographic order $x \succ y$), the problem becomes

$$y^2 - 3 = 0 \ \wedge \ -1 + x = 0 \ \wedge \ y > 0$$

and int can easily isolate solutions without a requirement of splitting (which is expensive as it increases the combinatorics of the problem).

However, as stressed in [4], Gröbner bases computation may require too much memory and be very time consuming compared to the speed-up they introduce. Thus, in [4] the authors propose a trade-off between pruning and computation time: gb is applied on subsets of the initial CSP, and the union of the resulting bases and the constraints that are not rewritten (such as inequalities, and equalities of non-polynomial expressions) forms the input of the propagation-based solver. We can describe this collaboration as follows:

$$\wedge\_\mathbf{p}(\mathbf{dc}(gb, \phi_=), \delta_{part}); int,$$

where δ_{part} is the $\wedge$**_separator** corresponding to the partitioning of the initial system introduced in [4].

7.2. The solver collaborations of CoSAc

CoSAc [23] is a constraint logic programming system for non-linear polynomial equalities and inequalities. The solving mechanism of CoSAc consists of five heterogeneous solvers working in a distributed environment and cooperating through a client/server architecture:

- *chr_lin* [14], implemented with CHRs, for solving linear constraints (equalities and inequalities),
- *gb* [13] for computing Gröbner bases, it is to be noticed that this solver is itself based on a client/server architecture,
- *maple_uni* for computing roots of a univariate polynomial equality, i.e., *maple_uni* extracts solutions from one equation, not from a set of equations,
- *maple_exp* for simplifying and transforming constraints (both this solver and the previous one are Maple [15] programs), and
- *ecl* for testing closed inequalities using ECL^{i}PS^e [20] features.

CoSAc uses several solving strategies, and thus, these solvers cooperate in three collaborations: S_{inc}, S_{fin} and S'_{fin}. We now focus on how these collaborations could be described in a simple way using our language. The collaborations of **CoSAc** are thus clarified: 1) every constraint cannot be treated by all the solvers, and using filters, we can make it clear and formalized; 2) distributed applications are implicit and form a part of the primitive semantics; 3) it becomes clear where improvements/strategies can be integrated.

S_{inc} is the incremental (in the sense of **CoSAc**) collaboration, i.e., it is applied as soon as a new constraint is added to the store. *maple_exp* transforms (e.g., expands polynomials and simplifies arithmetic expressions) all constraints so *eq_lin* can propagate information and simplify the set of linear equations (equalities and inequalities) filtered by $\phi_{=,<,lin}$:

$$S_{inc} = maple\_exp \;;\; \mathbf{dc}(eq\_lin, \phi_{=,<,lin}).$$

S_{fin} is one of the final solvers of **CoSAc**. It is applied once to the remaining constraints. First, constraints are simplified again by *maple_exp*, since S_{inc} may transform constraints in a syntax *gb* cannot understand. After computing Gröbner bases of the set of non-linear polynomial equalities (filtered by $\phi_=$), variables are eliminated (by *maple_uni*) one by one from univariate polynomials (filtered by $\phi_{=,uni}$), solutions are propagated, and linearized equations are solved (*eq_lin*). This process terminates when all variable have been eliminated or when there is no more univariate polynomial:

$$\begin{aligned} S_{fin} = \;& maple\_exp \;;\; \mathbf{dc}(gb, \phi_=) \;; \\ & \mathbf{dc}(maple\_uni, \phi_{=,uni}); \mathbf{dc}(eq\_lin, \phi_{=,<,lin})^\star. \end{aligned}$$

Here, we can see the flexibility and the simplicity of our control language. In **CoSAc**, the S_{fin} collaboration is fixed. From its description in our language, we can notice that *maple_uni* is applied by a *don't care* primitive. Some strategies can easily be introduced to improve the collaboration. In fact, *maple_uni* could be applied with a "best" primitive, ordering possible candidates with respect to the increasing degree of univariate polynomial equations (with a $\preceq_{degree}$ sorter). Using $\mathbf{best}(maple\_uni, \preceq_{degree}, \phi_{=,uni})$, variables could be eliminated from the lower degree equations first, and thus less arithmetic errors/roundings could be propagated to the system (and that is a weak point of **CoSAc**). Concerning *gb* and *eq_lin*, a "best" primitive would not help since these solvers consider the "maximal" set of filtered constraints.

S'_{fin} is an alternative to S_{fin} which is more efficient when eliminations of non-linear variables do not linearize any other constraint and only ground inequalities have to be checked by *ecl*:

$$\begin{aligned} S'_{fin} = \quad & maple\_exp \ ; \ \mathbf{dc}(gb, \phi_{=}) \ ; \\ & \mathbf{dc}(maple\_uni, \phi_{=,uni})^{\star} \ ; \ \mathbf{dc}(ecl, \phi_{<,ground})^{\star}. \end{aligned}$$

Again, better strategies can be introduced in **CoSAc**, since ground inequalities can be checked simultaneously. Using δ_{one}, a $\wedge$**_separator** that splits a set of n constraints into n singletons of atomic constraints, the application of *ecl* is improved:

$$\wedge\_\mathbf{p}(\mathbf{dc}(ecl, \phi_{<,ground}), \delta_{one}).$$

Note that we still need a filter for *ecl*, since δ_{one} does not perform any filtering.

As mentioned in [22], the first solvers of S_{fin} and S'_{fin} can be "factorized":

$$\begin{aligned} S''_{fin} = & maple\_exp \ ; \ \mathbf{dc}(gb, \phi_{=}) \ ; \\ & \quad \mathbf{pcc}(\ first, \\ & \qquad [(\mathbf{dc}(maple\_uni, \phi_{=,uni}); \mathbf{dc}(eq\_lin, \phi_{=,<,lin}))^{\star} \ , None, Id], \\ & \qquad [\mathbf{dc}(maple\_uni, \phi_{=,uni})^{\star} \ ; \mathbf{dc}(ecl, \phi_{<,ground}))^{\star}, None, Id]. \end{aligned}$$

The remaining parts of the collaborations are executed concurrently. No filtering is needed (Id for both sub-collaborations), and thus we do not have any sorter ($None$), since there is only one candidate after filtering, i.e., the initial set of constraints. We do not impose any property on the result, and we are interested in the sub-collaboration that will be faster (*first* property). Note that improvements for applying *ecl* and *maple_uni* still hold in S''_{fin}.

7.3. Combining consistencies

Box consistency [3] is a local consistency notion for interval constraints that relies on bounds of domains of variables: it is generally implemented as a (local) splitting of domains combined with the interval Newton method for determining consistent bounds of an interval. Hull consistency is another notion of consistency, stronger than box consistency. However, it can only be applied on primitive constraints that are either part of the original CSP, or are obtained by decomposing the constraints of the CSP. Then, the reduction of the "decomposed" CSP is weaker, but also faster. The idea of [3] is to combine these to consistencies in order to reduce the computation time for enforcing box consistency.

Let us consider $Hull$ and Box, two solvers that respectively enforce hull and box consistency of a CSP. Then, the combination of [3] can be described by

$$(HullC \ ; \ BoxC)^{\star}.$$

Since we can define both solvers and collaborations in our language, we now specify the $HullC$ and $BoxC$ solvers:

$$BoxC = \mathbf{dc}(Boxc, \phi_{\neg p})^\star \quad \text{and} \quad HullC = \mathbf{dc}(Hullc, \phi_p)^\star,$$

where ϕ_p (respectively, $\phi_{\neg p}$) filters one primitive (respectively non-primitive) constraint together with the domain constraints (e.g., $x \in [a, b]$) associated with each of its variables[9], $Boxc$ (respectively $Hullc$) is a component solver that, given a constraint c, enforces box (respectively, hull) consistency of c w.r.t. each of its variables.

We can also consider some inner strategies, such as reducing the variable with the largest domain. Then, $Hull$ and Box are defined as follows:

$$BoxC = \mathbf{best}(Boxc, \succeq_{Dom}, \phi_{\neg p})^\star \qquad HullC = \mathbf{best}(Hullc, \succeq_{Dom}, \phi_p)^\star,$$

where "$\succeq_{Dom}$" selects the candidate with the largest domain (see the sorter of Example 4).

Note that we could once again decompose these solvers into solvers that enforce box (or hull) consistency of one constraint with respect to one variable. Describing these solvers at this level, we are close to the generic propagation-based solver presented in Section 5: only the filter is different. Thus, we could imagine a more generic solver where the filter would also be a parameter. Then, solvers presented in Section 5 and in this section would be designed using the same pattern of operators of our language.

Note also that $(Hull\ ;\ Box)^\star$ can represent the solver int considered in Section 7.1. We could also think about some other description of $Hull$ and Box (e.g., using parallel application of solvers), but then we would not respect anymore the original combination of [3].

8. Conclusion

We have presented a strategy language for solving constraint satisfaction problems using solvers and collaboration of solvers. A key point in this work is the introduction of the concepts of constraint filters, separators, and sorters. These notions allow one to manage constraints with high-level mechanisms. Furthermore, they help describing syntactical transformations and manipulations generally hidden in the implementation of the current solvers. These concepts are then used to define strategy operators for applying solvers. These operators allow us to design solvers by combining the basic functions, and collaborations of solvers by combining the component solvers. This language can be seen as a Lego game, where bricks are basic solvers. These bricks are used to design more complex solvers and collaborations. They can be re-used, assembled together through strategies, used in

[9] ϕ_p is similar to $\phi_{D \wedge c \wedge Ds}$ (see Example 3) except that atomic constraints are forced to be primitive constraints.

higher collaborations, ... Patterns of solvers and strategies (i.e., assembling of operators) can be instantiated for different domains of constraints and different strategies of resolution.

The language is illustrated by several examples of constraints of different types and by defining solvers of different nature, such as well-known techniques for solving CSPs over finite domains and non-linear constraints over real domains, a generic propagation-based solver, optimization problems, collaboration of solvers (symbolic-numeric cooperation, simulation of **CoSAc**, combination of local consistencies). For each example, we have discussed standard strategies and proposed new strategies that clarify the use of our language. For lack of space, we did not present other solvers that we have already designed using our language, such as Gaussian elimination (and some standard strategies), and Gröbner bases computation.

We are currently working on the implementation of this language in order to evaluate the real applicability of this framework. We are confident that such a language can help exploring and testing new strategies. From a more theoretical point of view, we consider as further work verification of the termination properties of the strategy operators.

References

[1] K. R. Apt. *The Rough Guide to Constraint Propagation.* In J. Jaffar, editor, *Proc. of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1999. Invited lecture.

[2] F. Arbab and E. Monfroy. Heterogeneous distributed cooperative constraint solving using coordination. *ACM Applied Computing Review*, 6:4–17, 1999.

[3] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proceedings of International Conference on Logic Programming*, pages 230–244, Las Cruces, USA, 1999. The MIT Press.

[4] F. Benhamou and L. Granvilliers. Combining Local Consistency, Symbolic Rewriting, and Interval Methods. In *Proceedings of AISMC3*, volume 1138 of *Lecture Notes in Computer Science*, pages 144–159, Steyr, Austria, 1996. Springer-Verlag.

[5] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, March 1997.

[6] Henri Beringer and Bruno DeBacker. Combinatorial Problem Solving in Constraint Logic Programming with Cooperative Solvers. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North Holland, 1995.

[7] Alexander Bockmayr and Thomas Kasper. A unifying framework for integer and finite domain constraint programming. Research Report MPI-I-97-2-008, Max Planck Institut für Informatik, Saarbrücken, Germany, August 1997.

[8] B. Buchberger. Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose Ed., editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, Dordrecht - Boston - Lancaster, 1985.

[9] C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proceedings of Andrei Ershov Third International Conference Perspective of System Informatics, PSI'99*, volume 1755 of *Lecture Notes in Computer Science*, pages 402–415, Novosibirsk, Akademgorodok, Russia, 2000. Springer-Verlag.

[10] C. Castro and E. Monfroy. Basic Operators for Solving Constraints via Collaboration of Solvers. In *Proceedings of The Fith International Conference on Artificial Intelligence and Symbolic Computation, AISC'2000*, Lecture Notes in Artificial Intelligence, Madrid, Spain, 2000. Springer-Verlag. To Appear.

[11] Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, June 1998.

[12] Carlos Castro. COLETTE, Prototyping CSP Solvers Using a Rule-Based Language. In Jacques Calmet and Jan Plaza, editors, *Proceedings of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, AISC'98*, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 107–119, Plattsburgh, NY, USA, September 1998. Springer-Verlag.

[13] J-C. Faugere. *Résolution des systèmes d'équations algébriques.* PhD thesis, Université Paris 6, France, 1994.

[14] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[15] K. Geddes, G. Gonnet, and B. Leong. *Maple V: Language reference manual.* Springer Verlag, New York, Berlin, Paris, 1991.

[16] L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-Interval Cooperation in Constraint Programming. In *Proceedings of the 26th International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 150–166, University of Western Ontario, London, Ontario, Canada, 2001. ACM Press.

[17] Robert M. Haralick and Gordon L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[18] Alan K. Mackworth. Constraint Satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992. Second Edition.

[19] Philippe Marti and Michel Rueher. A Distribuited Cooperating Constraints Solving System. *International Journal of Artificial Intelligence Tools*, 4(1-2):93–113, 1995.

[20] M. Meier and J. Schimpf. ECLiPSe User Manual. Technical Report ECRC-93-6, ECRC (European Computer-industry Research Centre), Munich, Germany, 1993.

[21] E. Monfroy. An environment for designing/executing constraint solver collaborations. *ENTCS (Electronic Notes in Theoretical Computer Science)*, 16(1), 1998.

[22] E. Monfroy. The Constraint Solver Collaboration Language of BALI. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 211–230. Research Studies Press/Wiley, 2000.

[23] E. Monfroy, M. Rusinowitch, and R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of ACM Symposium on Applied Computing (SAC'96), Philadelphia, PA, USA*, pages 63–72. ACM Press, February 1996.

[24] Eric Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, France, November 1996. Also available in english and on-line at: http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/.

[25] Christophe Ringeissen. Cooperation of decision procedures for the satisfiability problem. In Franz Baader and Klaus Schulz, editors, *Proceedings of The First International Workshop Frontiers of Combining Systems, FroCoS'96*, pages 121–139. Kluwer Academic Publishers, 1996.

[26] Gert Smolka. Problem Solving with Constraints and Programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section.

[27] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[28] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2), 1997.