Basic Operators for Solving Constraints via Collaboration of Solvers

Carlos Castro[1] and Eric Monfroy[2]

[1] Departamento de Informática, Universidad Técnica Federico Santa María
Avenida España 1680, Casilla 110-V, Valparaíso, Chile
ccastro@inf.utfsm.cl
[2] Centrum voor Wiskunde en Informatica, CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Eric.Monfroy@cwi.nl

Abstract. In this paper, we propose a strategy language for designing schemes of constraint solver collaborations: a set of strategy operators enables one to design several kinds of collaborations. We exemplify the use of this language by describing some well known techniques for solving constraints over finite domains and non-linear constraints over real numbers via collaboration of solvers.

1 Introduction

In constraint programming, the programming process consists of formulating problems with constraints. Solutions of these so called Constraint Satisfaction Problems (CSPs) are generated by solvers. Numerous algorithms have been developed for solving CSPs and the resulting technology has been successfully applied for solving real-life problems. The design and implementation of these constraint solvers is generally an expensive and tedious task. Thus, the idea of reusing existing solvers is very interesting, but it also implies that we must have some tools to integrate them. Even more important, considering that some problems cannot be tackled or efficiently solved with a single solver, we definitively realize the interest of integrating and making cooperate several solvers [19, 4, 13, 20, 18]. This is called collaboration of solvers [15]. In order to make solvers collaborate, the need of powerful strategy languages to control their integration and application has been well recognized [16, 17, 1].

The existing approaches are generally not generic: they consider fixed domains (linear constraints [4], non-linear constraints over real numbers [18, 13, 3]), fixed strategies, or fixed scheme of collaboration (sequential [18, 3], asynchronous [13]). 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 [17]. In the system COLETTE [7, 8], a solver is viewed as a strategy that specifies the order of application of elementary operations expressed by transformation rules.

J.A. Campbell and E. Roanes-Lozano (Eds.): AISC 2000, LNAI 1930, pp. 142–156, 2001.

Extending ideas of **BALI** and COLETTE, we consider collaborations of solvers as strategies that specify the order of application of component solvers. In [9], we propose a strategy language for designing component or elementary constraint solvers and we exemplify its use by specifying several solvers (such as solvers for constraints over finite domains and real numbers). In this paper, we present the application of our language for prototyping constraint solving schemes via collaboration of solvers.

The main motivation for this work is to propose a general framework in which one can design 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 component solvers and designing collaborations of several solvers at the same level.

This paper is organized as follows: Section 2 presents basic definitions and notations. In Section 3, we present an overview of our strategy language whereas in Section 4 we detail its basic operators. In Section 5, we use our language for solving constraints over finite domains and real numbers via the collaboration of several solvers. Finally, we conclude in Section 6.

2 Definitions

Definition 1 (Constraint Systems and Constraint Solvers). *A constraint system is a 4-tuple* $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ *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.*

We denote by $\bot$ the *unsatisfiable* constraint and the *true* constraint 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 $ASS_{\mathcal{D}}^{\mathcal{V}}$. An assignment α extends uniquely to an 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 $\alpha \in ASS_{\mathcal{D}}^{\mathcal{V}}$ such that $\underline{\alpha}(c)$ holds. A constraint c is valid in $\mathcal{D}$ (denoted by $\mathcal{D} \models c$) if $Sol_{\mathcal{D}}(c) = ASS_{\mathcal{D}}^{\mathcal{V}}$. We use $\mathcal{V}ar(c)$ to denote the set of variables from $\mathcal{V}$ occurring in the constraint c.

Given a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, a *solver* is a computable function $S : \mathcal{L} \to \mathcal{L}$ satisfying the correctness and completeness properties, i.e., $\forall C \in \mathcal{L}$, $Sol_{\mathcal{D}}(S(C)) \subseteq Sol_{\mathcal{D}}(C)$ and $Sol_{\mathcal{D}}(C) \subseteq Sol_{\mathcal{D}}(S(C))$. 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 solved form with respect to S if $S(C) = C$.

In order to be able to manipulate specific parts of a constraint, we introduce the notions of *syntactical form* and *sub-constraint.*

Definition 2 (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. $Atom(C)$ denotes the set of atomic constraints that occur in C: $\{c | c \in \mathcal{L}_{At}$ and $C_{[c]}\}$.

Finally, in order to explicitly handle sub-parts of a constraint, we define the notions of *filter* to select specific parts of a constraint, and *sorter* to classify the elements of a list w.r.t. a given order [3].

Definition 3 (Filters and Sorters). *Given a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, a filter ϕ is a computable function $\phi : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{L} \times \mathcal{L})$ such that $\phi(C) = \{(Cf_i, C_i), \ldots, (Cf_n, C_n)\}$ for all $C \in \mathcal{L}$, where each C_i is a syntactical form of C and Cf_i is a sub-constraint of C_i.*

A sorter *Sorter, w.r.t. a partial order $\preceq$, 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. $\forall j \in [1, \ldots, n-1], Cf_j \preceq Cf_{j+1}$

The elements of $\phi(C)$ are called *candidates*. We define the filter Id which returns the initial set of constraints and the order $None$ which returns the initial list of candidates. Considering the filters ϕ_1 and ϕ_2 on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, then $\phi_1; \phi_2$ defined by $\phi_1(C) \cap \phi_2(C)$ is also a filter on $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ for all $C \in \mathcal{L}$.

3 An Overview of the Strategy Language

Most of the application mechanisms that we use in our strategy language are based on the same technique when applied to a constraint C:

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

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

[3] These transformations are normally hidden in existing solvers. In [9], we detail examples of the definition of filters and sorters.

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 are replaced in their corresponding (w.r.t. candidates) syntactical form of C.

The idea behind this scheme can be better understood in the following example. Suppose we are given the CSP over finite domains:

$$x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y$$

In order to find a solution we can carry out enumeration as follows:

- We first filter domain constraints in order to obtain a set of candidates:

$$\{(x \in [1,\ldots,10], \ \ x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y),$$
$$(y \in [1,\ldots,5], \ \ x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y)\}$$

- If we want to use the minimum domain criterion, a sorter will return the following sorted list of candidates:

$$[(y \in [1,\ldots,5], \ \ x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y),$$
$$(x \in [1,\ldots,10], \ \ x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y)]$$

- Applying a solver to split the "best" domain constraint we obtain:

$$y \in [1,\ldots,2] \ \vee \ y \in [3,\ldots,5], \ \ x \in [1,\ldots,10] \ \wedge \ y \in [1,\ldots,5] \ \wedge \ x \geq y$$

- After replacing the original constraint in the corresponding syntactical form we finally obtain:

$$x \in [1,\ldots,10] \ \wedge \ (y \in [1,\ldots,2] \ \vee \ y \in [3,\ldots,5]) \ \wedge \ x \geq y$$

This syntactical form is equivalent to the original set of constraints and once we activate operators properties we could continue the solving process.

4 The Strategy Language

Now we briefly present several *application mechanisms* to apply solvers to constraints. We assume that a solver is applied only once to a given set of constraints. In the following, we consider given a constraint system $CS = (\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$, solvers $S_1,\ldots,S_n$, filters $\phi_1,\ldots,\phi_n$, and partial orders $\preceq_1,\ldots,\preceq_n$.

We also use the notion of separators that are mainly defined to manipulate elements of conjunctions and disjunctions of constraints as elements of lists. A $\wedge$-***separator*** $\delta_\wedge$ is a function $\delta_\wedge : \mathcal{L} \rightarrow \mathcal{LC}$ s.t.: $\forall \ C \in \mathcal{L}, \exists n \in \mathbb{N}, \delta_\wedge(C) = [C_1,\ldots,C_n]$ where $C \approx C_1 \wedge \ldots \wedge C_n$. Similarly, a $\vee$-***separator*** $\delta_\vee$ is a function

$\delta_\vee : \mathcal{L} \rightarrow \mathcal{LC}$ such that: $\forall\ C \in \mathcal{L}$, $\exists n \in \mathbb{N}$, $\delta_\vee(C) = [C_1, \ldots, C_n]$ where $C \approx C_1 \vee \ldots \vee C_n$.

Finally, we use the notion of a constraint property p on a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ which is a function from constraints to Booleans (i.e., $p : \mathcal{L} \rightarrow \mathcal{B}oolean$).

We use five basic operators that are analogous to function compositions and that allow to design solvers by combining "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 1. 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.

We also use high level operators: two operators to apply a solver to specific components of a constraint, two operators to apply several solvers on a constraint, and two operators to apply a solver on each component of a conjunction or disjunction of constraints. Note that in the following, substitutions apply to every occurrence of sub-constraints.

$\mathbf{dc}(S_i, \phi)(C)$: this 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). For all $C \in \mathcal{L}, \mathbf{dc}(S_i, \phi)(C) = C'$, 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 $C' = C$.

$\mathbf{best}(S_i, \preceq, \phi)(C)$: this 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. For all $C \in \mathcal{L}, \mathbf{best}(S_i, \preceq, \phi)(C) = C'$, where:

- $[(Cf_1, C_1), \ldots, (Cf_n, C_n)] = Sorter(\preceq, \phi(C))$
- 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 $C' = C$.

$\mathbf{pcc}(p, (S_1, \preceq_1, \phi_1), \ldots, (S_n, \preceq_n, \phi_n))(C)$: this operator applies once one of the solvers S_i and returns a constraint that verifies the property p. For all $C \in \mathcal{L}, \mathbf{pcc}(p, [S_1, \preceq_1, \phi_1], \ldots, [S_n, \preceq_n, \phi_n])(C) = C'$, 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 $C' = C$.

$\mathbf{bp}((S_1, \preceq_1, \phi_1),\ldots,(S_n, \preceq_n, \phi_n))(C)$: this operator applies n solvers $S_1,\ldots,S_n$ on n sub-constraints of one syntactical form of the constraint. For all $C \in \mathcal{L}$, $\mathbf{bp}([S_1, \preceq_1, \phi_1],\ldots,[S_n, \preceq_n, \phi_n])(C) = C'$, where[4]:

- 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 $\sigma = \bigcup_{i\in[1,\ldots,n]} \sigma_i$.

$\wedge\_\mathbf{p}(S_i, \delta_\wedge)(C)$: this 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. For all $C \in \mathcal{L}, \wedge\_\mathbf{p}(S_i, \delta_\wedge)(C) = C'$, where:

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

$\vee\_\mathbf{p}(S_i, \delta_\vee)(C)$: this operator is analogous to $\wedge\_\mathbf{p}$ but $\delta_\vee$ determines disjuncts, and the final result is the disjunction of the results computed in parallel. For all $C \in \mathcal{L}, \vee\_\mathbf{p}(S_i, \delta_\vee)(C) = C'$, where:

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

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

Property 2. 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, separators $\delta_\wedge$ and $\delta_\vee$. Then, $\mathbf{dc}(S_i, \phi)$, $\mathbf{best}(S_i, \preceq, \phi)$, $\mathbf{pcc}(p, (S_1, \preceq_1, \phi_1),\ldots,(S_n, \preceq_n, \phi_n))$, $\mathbf{bp}((S_1, \preceq_1, \phi_1),\ldots,(S_n, \preceq_n, \phi_n))$, $\wedge\_\mathbf{p}(S_i, \delta_\wedge)$, and $\vee\_\mathbf{p}(S_i, \delta_\vee)$ are solvers.

5 Some Examples of Solver Collaborations

In this section we exemplify the use of our strategy language specifying solvers for constraints over finite domains and real numbers.

[4] Here we need the list of filters $[\phi_1,\ldots,\phi_n]$ to be stable and pairwise disjoint.

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_{x_i} being a finite set of values.

Solving this kind of problem can be seen as an interleaving process between local consistency verification and enumeration. The most widely used level of consistency verification, *Arc-Consistency*, can be expressed as the repeated application of the following transformation rule that reduces the set of possible values the variables can take.

$$x_i \in D_{x_i} \wedge c \wedge C \implies x_i \in RD(x_i \in D_{x_i}, c) \wedge c \wedge C \text{ if } RD(x_i \in D_{x_i}, c) \neq D_{x_i}$$

where $RD(x_i \in D_{x_i}, c) = \{v_i \in D_{x_i} \mid (\exists v_1 \in D_{x_1}, \ldots, v_{i-1} \in D_{x_{i-1}}, v_{i+1} \in D_{x_{i+1}}, \ldots, v_n \in D_{x_n}) : c(v_1, \ldots, v_i, \ldots, v_n)\}$.

Then, we define the solver *LocalConsistency* which applies this rule. In order to carry out enumeration, we consider the solver *SplitDomain* which transforms a domain constraint into a disjunction of two domain constraints if the width of the original domain is greater than or equal to a "minimal" width ϵ. For finite domains, ϵ is generally set to 1. For all $c = X \in D_X$ from $\mathcal{L}$:

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

$$SplitDomain(c) = X \in D'_X \vee X \in D''_X$$

  where $D_X = D'_X \cup D''_X$ [5],
- otherwise, $SplitDomain(c) = c$.

In order to select domain constraints, we define the filter ϕ_D that returns all domain constraints of the form $X \in D_X$, where D_X specifies the values that the variable X can take.

We also define the filter $\phi_{D \wedge c \wedge Ds}$ that returns sub-constraints which are the conjunction of a domain constraint, an atomic constraint, and a conjunction of domain constraints, i.e. , an atomic constraint with all the domain constraints of the variables occurring in it.

Finally, we define the sorter $\preceq_{Dom}$ that returns the candidate whose domain constraint is the one with the minimum set of values.

Then, the solver $FullLookahead_{MinDom}$, which returns all solutions to a CSP over finite domains, is defined in the following way:

$$\begin{aligned} FullLookahead_{MinDom} = \; & \mathbf{dc}(LocalConsistency, \phi_{D \wedge c \wedge Ds})^*; \\ & (\mathbf{best}(SplitDomain, \preceq_{Dom}, \phi_D); \\ & \quad \mathbf{dc}(LocalConsistency, \phi_{D \wedge c \wedge Ds})^*)^* \end{aligned}$$

[5] We generally also enforce that $D'_X \cap D''_X = \emptyset$.

This heuristic firstly enforces local consistency. Then, it carries out an enumeration step on the variable with the minimum set of remaining values, followed again by local consistency verification. Local consistency verification is always carried out on the whole set of constraints.

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

$$\wedge\text{\_}\mathbf{p}(FullLookahead_{MinDom}, \delta_{Var})$$

In this way, we are solving several CSPs in parallel. The 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.

5.2 Optimization Problems over Finite Domains

Here, we 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 the set of constraints is satisfied [21]. The work of Bockmayr and Kasper [5] seems to be the best currently available reference that 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, all of that 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:

$$\langle P, f, lb, ub\rangle \;\rightarrow\; \langle P, f, lb, \alpha(f)\rangle \textbf{ if } \alpha \in Sol(C \wedge f <^? ub)$$

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 the CSPs, as it is needed by this approach, we could use the already defined solver $FullLookahead_{MinDom}$.

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

$$\langle P, f, lb, ub\rangle \;\rightarrow\; \langle P, f, lb, \alpha(f)\rangle \textbf{ if } \alpha \in Sol(C \wedge f <^? \tfrac{(lb+ub)}{2})$$
$$\langle P, f, lb, ub\rangle \;\rightarrow\; \langle P, f, \tfrac{(lb+ub)}{2}, ub\rangle \textbf{ if } lb \neq ub \textbf{ and } Sol(C \wedge f <^? \tfrac{(lb+ub)}{2}) = \emptyset$$

The first rule tries to find a new value for the upper bound ub and reduces, by at least one-half, the range of possible values of the function f each time a new solution is obtained[6]. The second rule similarly updates the lower bound lb in the opposite situation. We call this approach *binary splitting* and we define the solver $MinSplitting$ implementing it.

Concerning the behavior of these strategies, we can note that the strategy $MinSatToUnsat$ takes a lot of time for reaching the minimal value of f, when it is located too 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 to know where the optimal solution is located, an *a priori* choice between these approaches is not possible in the general case. In order to improve the performances of these two basic solvers, we could make them collaborate in order to profit from the advantages of both of them, and to avoid their drawbacks.

A first scheme of cooperation between the solvers $MinSatToUnsat$ and $MinSplitting$ is expressed by the strategy $SeqOpt$:

$$SeqOpt = (MinSatToUnsat; MinSplitting)^\star$$

Using the strategy $SeqOpt$ both solvers are executed sequentially. Its obvious disadvantage is leaving a solver inactive, while the other one is working. Moreover, due to the exponential complexity of the problem under consideration, the whole process could be blocked if one solver cannot find a solution. To avoid this situation, we can think of running them concurrently, updating the current solution as soon as a new one is available, and stopping the other solver.

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

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 the faster, that is why we use the *first* property[7]. Using this strategy, a solver never waits for a solution coming from the other one. In the extreme case that all solutions are read from the same elementary solver until the final solution is obtained, the performance of this new solver, $ParOpt$, is the same as if one of the elementary solvers runs independently.

5.3 Combining Symbolic Rewriting and Interval Methods

Here, we consider systems of non-linear equations, and two solvers. Gröbner bases computation [6] (i.e., the *gb* solver) transforms a set of multivariate polynomial equalities into a normal form from which solutions can be derived more

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

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

easily than from the initial set. The second solver, *int*, is a propagation-based numerical solver over the real numbers. 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 pre-processing 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 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:

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

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.

However, as stressed in [3], Gröbner bases computation may require too much memory and be very time-consuming compared to the speed-up they introduce. Thus, in [3] 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 of 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 [3].

5.4 The Solvers of CoSAc

CoSAc [18] 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* [11], implemented with CHRs, for solving linear constraints (equalities and inequalities),

- *gb* [10] for computing Gröbner bases (note 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 [12] programs), and
- *ecl* for testing closed inequalities using ECL^{i}PS^e [14] features.

Since **CoSAc** uses several solving strategies, 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 control language. The collaborations 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 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 simplify 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 into 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 each variable has 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 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*. We can write it as:

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

Again, strategies can be introduced 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:

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

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

As mentioned in [17], the first solvers of S_{fin} and S'_{fin} can be "factorized":

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

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 the faster (*first* property). Note that improvements for applying *ecl* and *maple_uni* still hold in S''_{fin}.

5.5 Combining Consistencies

Box consistency [2] 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 intervals. 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 [2] is to combine these two 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 [2] can be described by:

$$(Hull \; ; \; Box)^{\star}$$

Since we can define both solvers and collaboration in our language, we now specify the *Hull* and *Box* solvers:

$$Box = (\mathbf{dc}(box, \phi_{\neg p}))^\star \text{ and } Hull = (\mathbf{dc}(hull, \phi_p))^\star$$

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, *box* (respectively *hull*) 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:

$$Box = (\mathbf{best}(box, \gg, \phi_{\neg p}))^\star \text{ and } Hull = (\mathbf{best}(hull, \gg, \phi_p))^\star$$

where "$\gg$" selects the candidate with the largest domain.

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. Note also that $(Hull\ ;\ Box)^\star$ can represent the solver *int* considered in Section 5.3. 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 [2].

6 Conclusions

We have presented a strategy language for solving CSPs via collaboration of solvers. A key point in this work is the introduction of basic strategy operators that allow the design of solvers by combining basic functions as well as the collaboration of solvers by combining component solvers. We have exemplified the use of this language by the simulation of well-known techniques for solving CSPs over finite domains and non-linear constraints over real domains. To show the broad scope of our control language's potential applications, we have designed several solvers that are considered of different nature (such as propagation based solvers, optimization over finite domain, and Gröbner bases computation). We are currently working on the implementation of this language in order to evaluate the real applicability of this framework. From a more theoretical point of view, we are considering as further work the verification of the termination properties of the strategy operators.

7 Acknowledgments

We are grateful to the anonymous referees who pointed out very accurate issues that allowed us to improve our work and the quality of this paper.

References

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

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

[3] F. Benhamou and L. Granvilliers. Combining Local Consistency, Symbolic Rewriting, and Interval Methods. In *Proc. of AISMC3*, volume 1138 of *LNCS*, pages 144–159, Steyr, Austria, 1996. Springer-Verlag.

[4] H. Beringer and B. DeBacker. Combinatorial Problem Solving in Constraint Logic Programming with Cooperative Solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North Holland, 1995.

[5] A. Bockmayr and T. 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, Aug. 1997.

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

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

[8] C. Castro. COLETTE, Prototyping CSP Solvers Using a Rule-Based Language. In J. Calmet and J. Plaza, editors, *Proc. of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, AISC'98*, volume 1476 of *LNCS*, pages 107–119, Plattsburgh, NY, USA, Sept. 1998. Springer-Verlag.

[9] C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proc. of Third International Conference Perspective of System Informatics, PSI'99*, volume 1755 of *LNCS*, pages 402–415, Novosibirsk, Russia, 2000. Springer-Verlag.

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

[11] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, 1995.

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

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

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

[15] E. Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes.* Phd thesis, Université Henri Poincaré - Nancy 1, France, Nov. 1996. Also available in english. On-line at: http://www.cwi.nl/~eric/Private/Publications/index.html.

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

[17] E. Monfroy. The Constraint Solver Collaboration Language of BALI. In D. 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.

[18] 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, *Proc. of ACM Symposium on Applied Computing (SAC'96), Philadelphia, PA, USA*, pages 63–72. ACM Press, February 1996.

[19] G. Nelson and D. C. Oppen. Simplifications by Cooperating Decision Procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[20] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Proc. of First Int. Workshop Frontiers of Combining Systems, FroCoS'96*, pages 121–139. Kluwer Academic Publishers, 1996.

[21] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.