Branching: the Essence of Constraint Solving

Antonio J. Fernández[1]* and Pat Hill[2]

[1] Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I.I., 29071 Teatinos, Málaga, Spain email:afdez@lcc.uma.es
[2] School of Computing, University of Leeds, Leeds, LS2 9JT, England email:hill@comp.leeds.ac.uk

Abstract This paper focuses on the branching process for solving any constraint satisfaction problem (CSP). A parametrised schema is proposed that (with suitable instantiations of the parameters) can solve CSP's on both finite and infinite domains. The paper presents a formal specification of the schema and a statement of a number of interesting properties that, subject to certain conditions, are satisfied by any instances of the schema. It is also shown that the operational procedures of many constraint systems (including cooperative systems) satisfy these conditions. Moreover, the schema is also used to solve the same CSP in different ways by means of different instantiations of its parameters.
Keywords: constraint solving, filtering, branching.

1 Introduction

To solve a *constraint satisfaction problem* (CSP), we need to find an assignment of values to the variables such that all constraints are satisfied. A CSP can have many solutions; usually either any one or all of the solutions must be found. However, sometimes, because of the cost of finding all solutions, *partial* CSP's are used where the aim is just to find the best solution within fixed resource bounds. An example of a partial CSP is a *constraint optimisation problem* (COP) that assigns a value to each solution and tries to find an optimal solution (with respect to these values) within a given time frame.

A common method for solving CSP's is to apply *filtering algorithms* (also called arc consistency algorithms or propagation algorithms) that remove inconsistent values from the initial domain of the variables that cannot be part of any solution. The results are propagated through the whole constraint set and the process is repeated until a stable set is obtained. However, filtering algorithms are, often, incomplete in the sense that they are not adequate for solving a CSP and, as consequence, it is necessary to employ some additional strategy called *constraint branching* that divides the variable domains and then continues with the propagation on each branch independently.

Constraint Solving algorithms have received intense study from many researchers, although the focus has been on developing new and more efficient

* This work was partly supported by EPSRC grants GR/L19515 and GR/M05645 and by CICYT grant TIC98-0445-C03-03.

methods to solve classical CSP's [8, 21] and partial CSP's [9, 15]. See [14, 17–19] for more information on constraint solving algorithms and [13, 16] for selected comparisons. To our knowledge, despite the fact that it is well known that branching step is a crucial process in complete constraint solving, papers concerned with the general principles of constraint solving algorithms have mainly focused on the filtering step [1, 7, 20].

In this paper, we propose a schema for constraint solving for both classical and partial CSP's that includes a generic formulation of the branching process. (This schema may be viewed as a generalisation and extension of the interval lattice-based constraint-solving framework in [7].) The schema can be used for most existing constraint domains (finite or continuous) and, as for the framework in [7], is also applicable to multiple domains and cooperative systems. We will show that the operational procedures of many constraint systems (including cooperative systems) satisfy these conditions.

The paper is organised as follows. Section 2 shows the basic notions used in the paper and Section 3 describes the main functions involved in constraint solving with special attention to those involved in the branching step. In Section 4 a generic schema for classical constraint solving is developed and its main properties are declared. Then, Section 5 extends the original schema for partial constraint solving and more properties are declared. Section 6 shows several instances of the schema to solve both different CSP's. Section 7 contains concluding remarks. Proofs of the properties can be found in a longer version of this paper available in http://www.lcc.uma.es/~afdez/recentpapers.

2 Basic concepts

Let $D, D_1, \ldots, D_n$ be sets or *domains*. Then $\#D$ denotes the cardinality of D, $\wp(D)$ its power set and $D_<$ denote any totally ordered domain. $\bot_D$ and $\top_D$ denote respectively, if they exist, the bottom and top element of D and fictitious bottom and top elements otherwise. Throughout the paper, Δ denotes a set of domains called *computation domains*.

Definition 1. *(Constraint satisfaction problem) A* Constraint satisfaction problem *(CSP) is a tuple* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ *where*

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is a non-empty finite set of variables.*
- $\mathcal{D} = \wp(D_1) \times \ldots \times \wp(D_n)$ *where* $D_i \in \Delta$.
- $\mathcal{C} \subseteq \wp(D_1, \ldots, D_n)$ *is a set of* constraints for $\mathcal{D}$.

If, as in the above definition, $\mathcal{D} = \wp(D_1) \times \ldots \times \wp(D_n)$, where $D_i \in \Delta$ for all $i \in \{1, \ldots, n\}$, then the set of all constraints for $\mathcal{D}$ is denoted as $\mathcal{C}_\mathcal{D}$ and the set $\{D_i \mid 1 \leq i \leq n\}$ is denoted as $\Delta_\mathcal{D}$.

Definition 2. *(Constraint store) Let* $S = (d_1, \ldots, d_n) \in \mathcal{D}$. *Then* S *is called a* constraint store for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. S *is* consistent *if, for all* $i \in \{1, \ldots, n\}$, $d_i \neq \emptyset$. S *is* divisible *if* S *is consistent and for some* $i \in \{1, \ldots, n\}$, $\#d_i > 1$. *Let*

$S' = (d'_1, \ldots, d'_n)$ *be another constraint store for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. *Then* $S \preceq_s S'$ *if and only if* $d_i \subseteq d'_i$ *for* $1 \leq i \leq n$.

S *is a* solution *for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ *if* $S = (\{s_1\}, \ldots, \{s_n\})$ *and* $(s_1, \ldots, s_n) \in c$, *for all* $c \in \mathcal{C}$. S' *is a* partial solution *for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ *if there exists a solution* S'' *for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ *such that* $S'' \prec_s S'$. *In this case we say that* S' covers S''.

The set of all solutions for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is denoted as $Sol(\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle)$. Note that, if $S \in Sol(\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle)$, then S is consistent and not divisible. If $(d_1, \ldots, d_n) \in \mathcal{D}$ and $i \in \{1 \ldots, n\}$, then $(d_1, \ldots, d_n)[d_i/d'] = (d_1, \ldots, d_{i-1}, d', d_{i+1}, \ldots, d_n)$.

Definition 3. *(Stacks) Let* $P = (S_1, \ldots, S_\ell) \in \wp(\mathcal{D})$. *Then* P *is a* stack for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. *Let* $P' = (S'_1, \ldots, S'_{\ell'})$ *be another stack for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. *Then* $P \preceq_p P'$ *if and only if for all* $S_i \in P$ $(1 \leq i \leq \ell)$, *there exists* $S'_j \in P'$ $(1 \leq j \leq \ell')$ *such that* $S_i \preceq_s S'_j$. *In this case we say that* P' covers P.

3 The Branching Process

This section describes the main functions used in the branching process.

First we define a filtering function which removes inconsistent values from the domains of a constraint store.

Definition 4. *(Filtering function)* $filtering_{\mathcal{D}} :: \wp(\mathcal{C}_{\mathcal{D}}) \times \mathcal{D} \to \mathcal{D}$ *is a called a* filtering function for $\mathcal{D}$ *if, for all* $S \in \mathcal{D}$,

(a) $filtering_{\mathcal{D}}(\mathcal{C}, S) \preceq_s S$;
(b) $\forall R \in Sol(\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle) : R \preceq_s S \implies R \preceq_s filtering_{\mathcal{D}}(\mathcal{C}, S)$.
(c) If $filtering_{\mathcal{D}}(\mathcal{C}, S)$ *is consistent and not divisible then* $filtering_{\mathcal{D}}(\mathcal{C}, S)$ *is a solution for* $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$.

Condition (a) ensures that the filtering never gains values, condition (b) guarantees that no solution covered by a constraint store is lost in the filtering process and condition (c) guarantees the correctness of the filtering function.

Variable ordering is an important step in constraint branching. We define a *selecting function* which provides a schematic heuristic for variable ordering.

Definition 5. *(Selecting function) Let* $S = (d_1, \ldots, d_n) \in \mathcal{D}$. *Then*

$$choose :: \{S \in \mathcal{D} \mid S \text{ is divisible}\} \to \{\wp(D) \mid D \in \Delta_{\mathcal{D}}\}$$

is called a selecting function for $\mathcal{D}$ *if* $choose(S) = d_j$ $(1 \leq j \leq n)$ *and* $\#d_j > 1$.

Example 1. Here is a naive strategy to select the left-most divisible domain.

$$\begin{aligned} \mathbf{P}&recondition : \{S = (d_1, \ldots, d_n) \in \mathcal{D} \text{ is divisible}\} \\ &choose_{naive}(S) = d \\ \mathbf{P}&ostcondition : \{\exists j \in \{1, \ldots, n\} \,.\, d = d_j \,,\ \#d_j > 1 \text{ and} \\ &\qquad \forall i \in \{1, \ldots, j-1\} : \#d_i = 1\}. \end{aligned}$$

In the process of branching, some computation domain has to be partitioned, in two or more parts, in order to introduce a choice point. We define a *splitting function* which provides a heuristic for value ordering.

Definition 6. *(Splitting function) Let $D \in \Delta$ and $k > 1$. Then*

$$split_D :: \wp(D) \rightarrow \underbrace{\wp(D) \times \ldots \times \wp(D)}_{k \ times}$$

is called a splitting function for D *if, for all $d \in \wp(D)$, $\#d > 1$, this function is defined $split_D(d) = (d_1, \ldots, d_k)$ such that the following properties hold:*

$$\begin{aligned} Completeness: &\quad d_1 \cup \ldots \cup d_k = d. \\ Disjointness: &\quad d_1 \cap \ldots \cap d_k = \emptyset. \\ Contractance: &\quad d_i \subset d,\ \forall i \in \{1, \ldots, k\}. \end{aligned}$$

To guarantee termination, even on continuous domains, an extension of the concept of precision map shown in [7] is applied here.

Definition 7. *(Precision map) Let $\Re\mathcal{I} = (\Re^+, Integer)$ where $\Re^+$ is the domain of non-negative reals. Then $precision_D$ is a* precision map *for $D \in \Delta$, if $precision_D$ is a strict monotonic function from $\wp(D)$ to $\Re\mathcal{I}$.*

Let $S = (d_1, \ldots, d_n)$ be a constraint store for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ and, for each $D \in \Delta_{\mathcal{D}}$, $precision_D$ is defined for D. Then, a precision map for $\mathcal{D} = (D_1, \ldots, D_n)$ *is defined as*

$$precision(S) = \sum_{1 \leq i \leq n} precision_{D_i}(d_i),$$

where the sum in $\Re\mathcal{I}$ is defined as $(a_1, a_2) + (b_1, b_2) = (a_1 + b_1, a_2 + b_2)$.

The monotonicity of the precision is a direct consequence of the definition.

Proposition 1. *Let S, S' be two constraint stores for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. If $S \prec_s S'$ then $precision(S) <_{\Re\mathcal{I}} precision(S')$.*

The precision map also means a novel way to normalise the selecting functions when the constraint system supports multiple domains. For instance, the well known *first fail principle* chooses the variable constrained with the smallest domain. For multiple domain constraint systems to emulate the first fail principle, we define $choose/1$ so that it selects the domain with the smallest precision[1]. We denote this procedure by $choose_{ff}$.

$\mathbf{P}recondition: \{S = (d_1, \ldots, d_n) \in \mathcal{D}$ is divisible$\}$
$\quad choose_{ff}(S) = d$
$\mathbf{P}ostcondition: \{\exists j \in \{1, \ldots, n\}\ .\ d = d_j\ , \#d_j > 1$ and
$\quad \forall i \in \{1, \ldots, n\} \backslash \{j\} : \#d_i > 1 \implies precision_{D_j}(d_j) \leq_{\Re\mathcal{I}} precision_{D_i}(d_i)\}$.

[1] It is straightforward to include more conditions e.g., if d_i, d_k, d_j have the same (minimum) precision, the most left domain can be chosen i.e., $d_{minimum(i,k,j)}$.

4 Branching in Constraint Solving

Figure 1 shows a generic schema for solving any CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$. This schema requires the following parameters: $\mathcal{C}$, the set of constraints to solve, a constraint store S for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, a bound $p \in \Re\mathcal{I}$ and a non-negative real bound ε. There are a number of values and subsidiary procedures that are assumed to be defined externally to the main branch procedure:

- a filtering function $\mathit{filtering}_{\mathcal{D}}/2$ for $\mathcal{D}$;
- a selecting function $\mathit{choose}/1$ for $\mathcal{D}$;
- a splitting function split_D for each domain $D \in \Delta_{\mathcal{D}}$;
- a precision map for $\mathcal{D}$ (therefore it is assumed that there is defined one precision map for each $D \in \Delta_{\mathcal{D}}$);
- a stack $P \in \wp(\mathcal{D})$ for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$.

It is assumed that all the external procedures have an implementation that terminates for all possible values.

$\mathit{procedure}\ \mathit{branch}(\mathcal{C}, S, p, \varepsilon)$
begin
  $S \leftarrow \mathit{filtering}_{\mathcal{D}}(\mathcal{C}, S);$ (1)
  if S is consistent then (2)
    if (S is not divisible or $p < \top_{\Re\mathcal{I}}$ and $p - \mathit{precision}(S) \leq (\varepsilon, 0)$) then (3)
      $\mathit{push}(P, S);$ %% Add S to top of P (4)
    else (5)
      $d_j \leftarrow \mathit{choose}(S);$ (6)
      $(d_{j1}, \ldots, d_{jk}) \leftarrow \mathit{split}_{D_j}(d_j),$ where $d_j \subseteq D_j;$ (7)
      $\left.\begin{array}{ll} \mathit{branch}(C, S[d_j/d_{j1}], \mathit{precision}(S), \varepsilon) & \vee \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots & \vee \\ \mathit{branch}(C, S[d_j/d_{jk}], \mathit{precision}(S), \varepsilon); & \end{array}\right\}$ %% Choice Points (8)
    $\mathit{endif};$
  $\mathit{endif};$
$\mathit{end}.$

Figure1. $\mathit{branch}/4$: A Generic Schema for Constraint Solving

Theorem 1. *(Properties of the branch/4 schema) Let S be the top element in $\mathcal{D}$ (i.e., $S = (D_1, ..., D_n)$), $\varepsilon \in \Re^+$ and $p = \top_{\Re\mathcal{I}}$. Then, the following properties are guaranteed:*

1. Termination*: if $\varepsilon > 0.0$ then $\mathit{branch}(\mathcal{C}, S, p, \varepsilon)$ terminates;*

2. Completeness*: if $\varepsilon = 0.0$ and the execution of $branch(\mathcal{C}, S, p, \varepsilon)$ terminates, then the final state for the stack P contains all the solutions for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$;*
3. Approximate completeness*: if $\varepsilon > 0.0$ and R is a solution for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, then an execution of $branch(\mathcal{C}, S, p, \varepsilon)$ will result in P containing either R or a partial solution R' that covers R.*
4. Correctness*: if $\varepsilon = 0.0$, the stack P is initially empty and the execution of $branch(\mathcal{C}, S, p, \varepsilon)$ terminates with R in the final state of P, then R is a solution for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$.*
5. Approximate correctness or control on the result precision*: If $P_{0.0}$, P_{ε_1} and P_{ε_2} are stacks resulting from any terminating execution of $branch(\mathcal{C}, S, p, \varepsilon)$ (where initially P is empty) when ε has the values $0.0, \varepsilon_1$ and ε_2, respectively, $0.0 < \varepsilon_1 < \varepsilon_2$ and $P_{0.0}$ is not empty, then $P_{0.0} \preceq_p P_{\varepsilon_1} \preceq_p P_{\varepsilon_2}$.*
   (In other words, the set of (possibly partial) solutions in the final state of the stack is dependent on the value of ε in the sense that lower ε is, closer to the real set of solutions is).

Observe that the bound ε guarantees termination and allows to control the precision of the results.

5 Solving optimisation problems

The schema in Figure 1 can be adapted to solve COPs by means of three new subsidiary functions.

Definition 8. *(Subsidiary functions and values) Let $D_<$ be a totally ordered domain[2]. Then we define*

- *a* cost function, *$fcost :: \mathcal{D} \to D_<$;*
- *an* ordering relation, *$\diamond :: D_< \times D_< \in \{>, <, =\}$;*
- *a* bound, *$\delta \in D_<$.*

Then the *extended schema*, $branch_+/4$, is obtained from the schema $branch/4$ by replacing Line 4 in Figure 1 with:

$$\text{if } fcost(S) \diamond \delta \text{ then } \delta \leftarrow fcost(S);\ push(P, S) \text{ endif;} \qquad (4^*)$$

Theorem 2. *(Properties of the $branch_+/4$ schema) Let S be the top element in $\mathcal{D}$ (i.e., $S = (D_1, ..., D_n)$), $\varepsilon \in \Re^+$ and $p = \top_{\Re\mathcal{I}}$. Then, the following properties hold:*

1. Termination*: if $\varepsilon > 0.0$, then the execution of $branch_+(\mathcal{C}, S, p, \varepsilon)$ terminates;*
2. *If fcost is a constant function with value δ and $\diamond$ is $=$, then all properties shown in Theorem 1 hold for the execution of $branch_+(\mathcal{C}, S, p, \varepsilon)$.*
3. Soundness on optimisation*: if $\varepsilon = 0.0$, $\diamond$ is $>$ (resp. $<$), $\delta = \bot_{D_<}$ (resp. $\top_{D_<}$), the stack P is initially empty and the execution of $branch_+(\mathcal{C}, S, p, \varepsilon)$ terminates with P non-empty, then the top element of P is the first solution found that maximises (resp. minimises) the cost function.*

[2] Normally $D_<$ would be $\Re$.

Unfortunately, if ε is higher than 0.0, we cannot guarantee that the top of the stack contains a solution or even a partial solution for the optimisation problem. However, by imposing a monotonicity condition on the cost function $fcost/1$, we can compare solutions.

Theorem 3. *(More properties on optimisation) Suppose that, for $i \in \{1,2\}$, P_{ε_i} is a stack resulting from the execution of $branch_+(\mathcal{C}, S, p, \varepsilon_i)$ where $\varepsilon_i \in \Re^+$. Suppose also that $top(P)$ returns the top element of a non empty stack P. Then, if $\varepsilon_1 < \varepsilon_2$ the following property hold.*

Approximate soundness*: If for $i \in \{1,2\}$, P_{ε_i} is not empty, and $top(P_{\varepsilon_2})$ is a solution or covers a solution for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, then, if $fcost/1$ is monotone and $\diamond$ is $<$ (i.e., a minimisation problem),*

$$fcost(top(P_{\varepsilon_1})) \preceq_{D_<} fcost(top(P_{\varepsilon_2})),$$

and, if $fcost/1$ is anti-monotone and $\diamond$ is $>$ (i.e.,a maximisation problem),

$$fcost(top(P_{\varepsilon_1})) \succeq_{D_<} fcost(top(P_{\varepsilon_2})).$$

Therefore, by using a(n) (anti-)monotone cost function, the lower ε is, the better the (probable) solution is. Moreover, decreasing ε is a means to discard approximate solutions. For instance, in a minimisation problem, if

$$fcost(top(P_{\varepsilon_1})) \succ_{D_<} fcost(top(P_{\varepsilon_2}))$$

with $fcost/1$ monotone, then, by the *approximate soundness* property it is deduced that $top(P_{\varepsilon_2})$ cannot be a solution or cover a solution.

6 Examples

To illustrate the schemas $branch/4$ and $branch_+/4$ presented in the previous two sections, several instances of $branch/4$ are given for some well-known domains of computation. In the following, $branch_X$ denotes an instance of the schema $branch/4$ for solving the CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where $X \subseteq \Delta_{\mathcal{D}}$. We assume that

$$\Delta = \{Bool, Integer, \Re, Set\ Integer\} \cup \{Interv(D) \mid D \text{ is a lattice}\}.$$

where $Interv(D)$ denotes the set $\{(d_1, d_2) \mid d_1, d_2 \in D, d_1 \leq d_2\}$.

To identify $branch_{\mathcal{D}}$, we indicate a possible definition for both the splitting function and the precision map for each $D \in \Delta_{\mathcal{D}}$ and assume that both a selecting function and a filtering function for $\mathcal{D}$ have been already defined. We also indicate the initial value of $S \in \mathcal{D}$, so that the execution of $branch_{\mathcal{D}}(\mathcal{C}, S, p, \varepsilon)$ allows to solve the CSP where $\varepsilon \in \Re^+$.

The finite domain (FD) Constraint solving in a FD of sparse elements is solved by an instance $branch_{FD}$ as defined below where $split_{FD}$ is defined as a naive enumeration strategy in which values are chosen from left to right. For

example, consider a finite domain of integers [5], Booleans [4] or finite sets of integers [6]).

$$FD \in \{Integer, Bool, Set\ Integer\},$$
$$branch_{FD} \begin{cases} S = (\underbrace{FD, \ldots, FD}_{n\ times}); \\ precision_{FD}(d) = (\#d, 0); \\ split_{FD}(\{a_1, a_2, a_3, \ldots, a_k\}) = (\{a_1\}, \{a_2, a_3, \ldots, a_k\}). \end{cases}$$

Finite closed intervals Many existing *FD* constraint systems solve constraints defined in the domain of closed intervals $[a, b]$ where $a, b \in FD$ and denoted here by $a..b$. Usually a, b are either integers [3], Booleans[3] [4] or finite sets of integers [10]. Here are two instances of our schema that solve CSP's on these domains:

$$FD \in \{Integer, Bool\},$$
$$branch_{Interv(FD)} \begin{cases} S = (\underbrace{\bot_{FD}..\top_{FD}, \ldots, \bot_{FD}..\top_{FD}}_{n\ times}); \\ precision_{Interv(FD)}(a..b) = (b - a, 0); \\ split_{Interv(Integer)}(a..b) = (a..a, a + 1..b). \end{cases}$$

$$FD = Set\ Int,$$
$$branch_{Interv(FD)} \begin{cases} S = (\underbrace{\emptyset..Integer, \ldots, \emptyset..Integer}_{n\ times}); \\ precision_{Interv(FD)}(a..b) = (\#b - \#a, 0); \\ split_{Interv(FD)}(a..b) = (a..b \backslash \{c\}, a \cup \{c\}..b) \text{ where } c \in b \backslash a. \end{cases}$$

Lattice (interval) domain In [7], we have described a generic filtering algorithm that propagates interval constraints on any domain L with lattice structure subject to the condition that a function $\circ_L :: L \times L \to \Re$ is defined that is strictly monotonic on its first argument and strictly anti-monotonic on its second argument. Below we provide an instance to solve any CSP defined on $Interv(L)$:

$$branch_{Interv(L)} \begin{cases} L \text{ is a lattice and } S = (\underbrace{[\bot_L, \top_L], \ldots, [\bot_L, \top_L]}_{n\ times}); \\ precision_{Interv(L)}(r) = \begin{cases} (b \circ_L a, 2) \text{ if } r = [a, b]; \\ (b \circ_L a, 1) \text{ if } r = (a, b]; \\ (b \circ_L a, 1) \text{ if } r = [a, b); \\ (b \circ_L a, 0) \text{ if } r = (a, b); \end{cases} \\ split_{Interv(L)}(\{a, b\}) = (\{a, c\}, (c, b\}) \text{ where } a \preceq_L c \prec_L b. \end{cases}$$

[3] The Boolean domain is considered as the integer subset $\{0, 1\}$.

$\{a, b\}$ denotes any interval in L. With this instance we have a constraint solving mechanism for solving (interval) constraints defined on any domain with lattice structure. Thus it is a good complement to the filtering algorithm in [7]. Note also that if L is $\Re$ and $\circ_L$ is $-$, we obtain the instance $branch_{Interv(\Re)}$ (also, if $c = \frac{b-a}{2.0}$ we have a usual strategy of real interval division at the mid point).

A cooperative domain The schema also supports cooperative instances that solve CSP's defined on multiple domains. This is done by mixing together several instances of the schema $branch/4$. As an example, consider $branch_{BNR}$ as defined below where $split_{Interv(D)}$ and $precision_{Interv(D)}$ are defined as in previous examples for $D \in \{Bool, Integer, \Re\}$:

$$branch_{BNR} \begin{cases} \Delta = \{Interv(D) \mid D \in \{Bool, Integer, \Re\}\}. \\ S = (\underbrace{\emptyset..D_1, \ldots, \emptyset..D_n}), \{D_1, \ldots, D_n\} \subseteq \{Bool, Integer, \Re\}. \end{cases}$$

This instance simulates the well known *splitsolve* method of CLP(BNR) [2].

7 Concluding remarks

This paper analyses the branching process in constraint solving. We have provided a generic schema for solving CSP's on finite or continuous domains as well on multiple domains. We have proved key properties such as correctness and completeness. We have shown how termination may be guaranteed by means of a *precision map*. We have also shown, by means of an example, how, for systems supporting multiple domains, the precision map can be used to normalise the heuristic for variable ordering.

By using a schematic formulation for the branching process, we have indicated which properties of main procedures involved in branching are responsible for the key properties of constraint solving.

By combining a filtering function satisfying our conditions with an appropriate instance of our schema, we obtain an operational semantics for a constraint programming domain (for example: FD, sets of integers, Booleans, multiple domains, ...,etc) and systems designed for constraint solving such as clp(FD) [3], clp(B) codognet+:local-prop-jar96, DecLic [11], clp(B/FD) [4], CLIP [12], Conjunto [10] or CLP(BNR) [2].

References

1. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
2. F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *The Journal of Logic Programming*, 32(1):1–24, July 1997.
3. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

4. P. Codognet and D. Diaz. A simple and efficient boolean solver for constraint logic programming. *The Journal of Automated Reasoning*, 17(1):97–129, 1996.
5. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In ICOT, editor, *International Conference on Fith Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, November-December 1988. OHMSHA Ltd. and Springer-Verlag.
6. A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {log}: A language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44, July 1996.
7. A.J. Fernández and P.M. Hill. An interval lattice-based constraint solving framework for lattices. In Aart Middeldorp and Taisuke Sato, editors, *4th International Symposium on Functional and Logic Programming (FLOPS'99)*, number 1722 in LNCS, pages 194–208, Tsukuba, Japan, November 1999. Springer Verlag.
8. E.C. Freuder and P. Hubbe. Extracting constraint satisfaction subproblems. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 548–557, Québec, Canada, August 1995. Morgan Kaufman.
9. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(21-70):21–70, 1992.
10. C. Gervet. Interval propagation to reason about sets: definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
11. F. Goualard, F. Benhamou, and L. Granvilliers. An extension of the WAM for hybrid interval solvers. *The Journal of Functional and Logic programming*, 1999(1):1–36, April 1999. Special issue of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.
12. T.J. Hickey. CLIP: A CLP(Intervals) Dialect for Metalevel Constraint Solving. In E. Pontelli and V.Santos Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in LNCS, pages 200–214, Boston, USA, 2000. Springer Verlag.
13. G. Kondrak and P. Van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89(1-2):365–387, January 1997.
14. V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, Spring 1992.
15. P. Meseguer and J. Larrosa. Constraint satisfaction as global optimization. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 579–585, Québec, Canada, August 1995. Morgan Kaufman.
16. B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
17. Z. Ruttkay. Constraint satisfaction-a survey. *CWI Quaterly*, 11(2-3):163–214, 1998.
18. B.M. Smith. A Tutorial on Constraint Programming. Research Report 95.14, University of Leeds, School of Computer Studies, England, April 1995.
19. P. Van Hentenryck. Constraint solving for combinatorial search problems: A tutorial. In U. Montanari and F. Rossi, editors, *1st International Conference on Principles and Practice of Constraint Programming (CP'95)*, number 976 in LNCS, pages 564–587, Cassis, France, 1995. Springer Verlag.
20. P. Van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.
21. R.J. Wallace. Why AC-3 is Almost Always Better than AC4 for Establishing Arc Consistency in CSPs. In R. Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 239–247, Chambéry, France, August-September 1993. Morgan Kaufmann.