A Visualization Tool for Constraint Program Debugging

Frédéric Goualard and Frédéric Benhamou
Institut de Recherche en Informatique de Nantes
2, rue de la Houssinière, B.P. 92208
F-44322 Nantes Cedex 3
Frederic.[Goualard,Benhamou]@irin.univ-nantes.fr

Abstract

Constraint programming is an emerging technology that offers an original approach allowing for efficient and flexible solving of complex problems. Its main advantage relies in its ability to compute with partial information expressed in terms of constraints. These constraints are monotonically accumulated during the program execution in order to restrict the problem search space. In this paper we address one of the cornerstones of this technology, namely the current lack of debugging facilities. In particular, visualization and understanding of the underlying constraint system during program executions is very important. We propose to structure this huge, flat and intricate part of the execution data in order to provide access to high level examination of its evolution. More precisely, we present a means to hierarchically organize sets of constraints in order to divide them into manageable parts while preserving computation correctness. Soundness of our method is shown, an algorithm supporting it is given, and an implemented prototype exhibiting its effectiveness is described.

1. Introduction

Constraint programming[5] is a language-independent paradigm permitting to solve a problem by simply specifying the properties its solution must verify. Resolving problems that way leads to programs that are both elegant and efficient. This declarativeness led many researchers to introduce it in different languages, be they imperative (REF-ARF [9]), object-oriented (ILOG Solver [16]), logic (CHIP [1], Prolog IV [4], clp(fd) [6]), or functional (HOPE [8]).

However, constraint programming (CP) is not as widely used in the industrial circles as its potential entitles it. The lack of programming environments offering advanced features for easily debugging and profiling constraint programs is certainly not extraneous to this situation. Actually, the expressiveness of constraint programming permits shortening programs at the cost of forbidding the use of traditional debuggers such as step-by-step tracers, since the main part of the solving process—that is, the addition of constraints into a constraint network called the store gathering all previously introduced constraints, and the re-invocation of all the constraints sharing some variables whose domain of possible values has been reduced by such additions—has no direct counterpart in the source code and is concealed from the programmer.

In this paper, we present a new abstraction of the store, called S-box—where the “S” stands for “store”—permitting to gather sets of constraints into one new “global” constraint which is the conjunction of the embedded constraints. The S-box abstraction allows us to device a new kind of CP debugger that discloses the store contents to the programmer in a usable way: a set $S$ of numerous constraints having a global semantics on its own may be represented in the store by a new constraint (S-box formed by the constraints in $S$), thus decreasing the store complexity. Moreover, since S-boxes may contain other S-boxes, the S-box abstraction leads to a hierarchically-structured store that may be explored in a handy way by the user focusing in and out the S-boxes he wants to inspect. S-box properties are such that bugs in a constraint program may be tracked down by isolating them into smaller and smaller sets of constraints put into suitable S-boxes. The semantics of the program is shown not to be affected by this debugging process though the re-invocation order of constraints is modified.

The rest of the paper is organized as follows: Section 2 briefly introduces some fundamental constraint programming notions, focusing on constraint logic programming, that is CP embedded in a logic language such as Prolog [7]; Section 3 describes the difficulties the programmer has to face when debugging constraint programs with traditional tools; Section 4 introduces the S-box notion permitting to structure hierarchically the constraint store; its soundness is shown, and an algorithm supporting it is given to replace the propagation algorithm of constraint solvers; Section 5
surveys the functionalities of a prototype for a S-box-based debugger; Lastly, Section 6 summarizes the paper’s contribution and points out some directions for future research.

2. Constraint logic programming in a nutshell

For the sake of simplicity, we will restrict ourselves in the following to Constraint Logic Programming (CLP) and to numerical constraints involving only integer-valued variables.

Consider the constraint \( c_1 : x + y = z \) where the initial variables’ domains are respectively \( D_x = \{2, 3, \ldots, 6\} \), \( D_y = \{2, \ldots, 4\} \), and \( D_z = \{3, \ldots, 5\} \). Some values of these domains are clearly inconsistent with \( c_1 \). For example, \( x \) cannot be equal to 6 since there is no value in \( D_y \) and \( D_z \) such that \( 6 + y = z \). Hence, this value may be eliminated from \( D_z \). A constraint narrowing operator [2] \( N[c]\) for a \( n \)-ary constraint \( c \) (shortened thereafter to CNO) is a contracting, correct and monotone function discarding the \( n \)-tuples from a Cartesian product of variables’ domains that are inconsistent for \( c \). This filtering of consistent values is related to local consistency notions [13]. In the example given above, applying the \( N[c_1] \) operator to the Cartesian product \( D_x \times D_y \times D_z \) would narrow the domains to \( D_x' = \{2, 3\} \), \( D_y' = \{2, 3\} \), and \( D_z' = \{4, 5\} \). The remaining values are such that for every \( \alpha \) in \( D_x' \) (resp. \( D_y' \) and \( D_z' \)), there exists at least one pair \( (\beta, \gamma) \) of consistent values in \( D_y' \times D_z' \) (resp. \( D_x' \times D_z' \) and \( D_x' \times D_y' \)) such that \( c_1(\alpha, \beta, \gamma) \) does hold (arc consistency [13]). If \( y \) is later bound to 3 (e.g., by adding the constraint \( c_2 : y \geq 3 \)), the \( N[c_1] \) operator will be able to further narrow down \( D_x' \) and \( D_z' \) to the final domains \( D_x'' = \{2\} \), \( D_y'' = \{5\} \). Consequently, constraints sharing variables have to be “linked” in some way, thereby permitting the re-invocation of all the constraints where occurs a variable whose domain has just been narrowed by one of them. A constraint store is such a network where nodes are constraints linked together whenever they share at least one variable (see example in Figure 1).

\[
\mathcal{S} = \begin{cases} 
  x^2 = y \\
  x + y = z \\
  z = k + y \\
  \end{cases}
\]

Figure 1. A constraint system and its associated store

Algorithm 1 describes in details the addition of constraints into the store: constraints \( c_1, \ldots, c_m \), are added into a propagation list (managed as a FIFO list in most implementations); their narrowing operators \( N[c_1], \ldots, N[c_m] \), are applied onto the Cartesian product of the variables’ domains, and all the constraints involving a variable whose domain has been narrowed are pushed into the propagation queue. The whole process terminates on failure (i.e. a domain is narrowed to the empty set) or on emptiness of the propagation queue (the common fixed-point included in the initial Cartesian product of domains of all the narrowing operators is reached).

\begin{algorithm}
\begin{algorithmic}
\Function{the narrowing algorithm}{
\textbf{in:} \{\( N[c_1], \ldots, N[c_m] \)}
\textbf{inout:} \textbf{D} = \textbf{D}_1 \times \cdots \times \textbf{D}_n
\begin{algorithmic}
\Function{NAR}{
\State \textbf{Q} := \{c_1, \ldots, c_m\} \quad \% Adding \( c_i \)s in the propagation list
\State \textbf{S} := \textbf{S} \cup \{c_1, \ldots, c_m\} \quad \% Adding \( c_i \)s in the store
\While{(\textbf{Q} \neq \emptyset \textbf{and} \textbf{D} \neq \emptyset)}
\State \( c := \text{choose a constraint} \ c_i \text{ in} \ \textbf{Q} \)
\State \textbf{D} := \textbf{D} \quad \% Adding \( c_i \)s in the store
\If{\textbf{D} \neq \emptyset}
\State \textbf{Q} := \textbf{Q} \cup \{c_i \in \mathcal{S} \mid \exists x_k \in \text{Var}(c_j) \land \textbf{D}_k \neq \emptyset\}
\State \textbf{D} := \textbf{D} \quad \% Adding \( c_i \)s in the store
\EndIf
\State \textbf{Q} := \textbf{Q} \setminus \{c\}
\EndWhile
\EndFunction
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

Narrowing operators are in general unable to discard all the inconsistent values from the Cartesian product of domains (partial consistency only). Hence, applying Algorithm 1 is often insufficient to achieve the overall consistency. Then, a labeling process—consisting in assigning to each variable all of its possible remaining values through backtracking—occurs when all constraints have been added to the store, thus permitting to isolate all the consistent assignments for the variables involved in the constraint system.

3. Arising difficulties when debugging constraint programs

In this section, the efficiency of the constraint programming paradigm is shown, and the drawbacks of the qualities permitting such an efficiency with respect to debugging are pointed out.

3.1. Constraint programming efficiency

Consider the following problem, solved thereafter in both an imperative language and a constraint programming language:

Example 1 (Yoshigahara problem [18]) Let \( A, \ldots, I \), be nine integer-valued variables. Use all digits 1 through 9
only once such that
\[
\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1
\]  
(1)

where $BC$, $EF$ and $HI$ stand for the number obtained by concatenating the digit values of the corresponding variables.

The Yoshigahara problem has only one solution (up to permutations of quotients): $5/34 + 7/68 + 9/12 = 1$.

A naïve resolution of this problem involves the generation of the $9^9 = 387,420,489$ possible assignments to variables $A$ through $I$, followed by the test whether each assignment meets the requirement (1). This is the so-called generate and test approach illustrated by the Java program given in Program 1.

**Program 1. Yoshigahara problem in Java**

```java
class Yoshigahara {
    private static boolean alldifferent(int a, ...) {
        // Implementation...
    }
    public static void main(String[] args) {
        // Generate
        // Test
    }
}
```

On the other hand, a constraint-based resolution starts stating all the constraints before generating remaining possible assignments. As a result, many assignments are not considered. For example, if the addition of a constraint leads to discarding the value 6 from the domain of the $A$ variable, all the 9-tuples of the form $(6, \ldots, \ldots, \ldots, \ldots)$ are no longer considered during the labeling phase. This is the test and generate approach illustrated by the DecLIC [10] program given in Program 2.

**Program 2. Yoshigahara problem in DecLIC**

```c
:- L=[A,B,C,D,E,F,G,H,I],
is_integer(L),
alldifferent(L),
A/(10*B+C)+D/(10*E+F)+G/(10*H+I) $= 1,
labelingff(L). % Generate/enumerate (smallest domain 1st)
```

On an AMD K6-166MHz based PC, all solutions are found by Program 1 in 1 778.2s, and by Program 2 in 1.83s., which shows the superiority of constraint programming for such kind of problems.

3.2. Drawbacks of CP expressiveness

Debugging Program 1 is easily done using traditional debuggers allowing to run it step-by-step since every instruction performed to solve the problem appears in the code. This is not the case for Program 2. The point is that the actual work is not reflected by the instructions appearing in it. For example, if we use a Prolog tracer to run it step-by-step from Line 4 to Line 6, we will only be able to see the domain modifications induced by introducing the constraint appearing in Line 5 into the store, and we will miss everything that was done when propagating intermediate modifications in the constraint network (re-invocation of constraints and intermediate modifications of domains). In particular, we get no information at all in case of failure.

Actually, there exist two independent levels in CLP programs:

1. the Prolog level (clausal structure) which may be debugged using the traditional debugging facilities;
2. and the store level (propagation of domain modifications) for which traditional debugging facilities are useless.

The store level is clearly unreachable from “traditional” debuggers. Hence, a specialized tool able to display the constraint store in a readable way has to be devised for this purpose.

4. Visualizing the store

The debugging tools integrated in CP languages such as ECLiPSe [14] and Oz [17] do not allow to visualize directly the store but display the variables’ domains and their modifications “in real time”. Though these are important and useful functionalities, we believe that a more direct presentation of the relations between the constraints in the store is mandatory to fully debug CP programs. This sounds particularly true when variables are real-valued, since then, domain modifications bring few information.

The idea is then to display a representation of the store as a constraint network such as the one appearing in Figure 1. However, displaying the store as it is manipulated by the solver (raw form) is useless since it is a huge, flat, collection of constraints with no structure at all—even for toy problems. Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (e.g. in Prolog IV), and stored constraints are not always those given by the user (e.g. solvers like clp(BNR) decompose constraints into some primitive constraints, using new variables).

It is then necessary to let the user control the size and complexity of the store.
4.1. Structuring the store

A CLP program is structured by clauses. This structure is lost when the constraints are added to the store whereas it often conveys some useful information concerning the meaning of some constraint sets. For example, let us consider Program $P$ [11] (Prog. 3) describing the problem of finding the collision point between a wall and a ball.

**Program 3. Code for Program $P$ with tagged Prolog goals**

```
object_A(X,Y,Z):- % Shape of the wall
  X ≤ 0, Y ≤ 0, Z ≤ 0.
object_B(X,Cx,Y,Cy,Z):- % Shape of the ball
  X(Cx)^2 + Y(Cy)^2 + Z(Cz)^2 = 1.
center_B(T,Cx,Cy,Cz):- % Pos. of the ball center at T
  T^2 - Cx = 10,
  2T - Cy = 10,
  Cz - T^2 + TT = 0.
obj ect_B_moving(T,X,Y,Z):-
  % Pos. of pt. (X,Y,Z) in ball at T
  center_B(T,Cx,Cy,Cz),
  X(Cx) = X - Cx,
  Y(Cy) = Y - Cy,
  Z(Cz) = Z - Cz,
  object_B(X(Cx),Y(Cy),Z(Cz)).
:- T > 0, object_A(X,Y,Z),
o bject_B_moving(T,X,Y,Z).
```

The `center_B(T,Cx,Cy,Cz)` clause, defining the center position of the ball, contains three constraints. It could be considered as a new user-defined “global constraint” whose semantics is quite intuitive. However, introducing these three constraints in the store discards the special link between them. Figure 2 displays the store for Program $P$. Note that it is already an abstraction of the actual store since all the constraints appearing in it are the constraints as given by the user (no decomposition into primitives, nor renaming of variables as usually done in Prolog); and yet, it may already be too complicated to be useful.

A first approach to reduce the store complexity is to reintroduce the structure induced by the program clauses. Figure 3 shows the store for $P$ where each clause is represented by a box (more precisely, the behaviour at runtime is as follows: a box is created whenever the control “enters” into a clause; added constraints are then put into that box until the control leaves the clause).

Structuring the store that way permits presenting abstractions of it by masking all constraints in a box. For example, the store of Figure 3 may be abstracted into an equivalent store containing only three constraints: $T \geq 0$, `object_A(X,Y,Z)`, and `object_B_moving(T,X,Y,Z)`—Fig. 4.
Note that the number of edges between “constraints” has dramatically decreased. It is then easier to add some guards onto these links warning the programmer of the violation of some property when propagating variables’ domains from a box to another. For example, assume that the propagation of some domains given by the user as an input to the box object_B.moving(\(T, X, Y, Z\)) does not lead to the right domains on output; in order to find what went wrong, the user can focus on the box of object_B.moving(\(T, X, Y, Z\)), obtaining the store described in Figure 5.

![Image of object_B.moving](center_B_T,Cx,Cy,Cz)

**Figure 5. The new store after focusing on object_B.moving(X,Y,Z)**

Focusing on a box \(\sigma\) means temporarily restricting the set of constraints in the store to the ones contained in \(\sigma\) and its sub-boxes. As a consequence, propagation of domain narrowing has to be done in the box having the focus only—otherwise, we cannot be assured that the bug is really in the box—and the focus must be reset to the entire store sooner or later during the debugging process, such that all constraints have a chance to be reconsidered, and the program semantics preserved.

A greater flexibility may be obtained by letting the programmer create boxes which are not directly inspired by the clausal structure of the program. This can be done by tagging all the constraints to be put in the same box (cf. Program 3, where tags correspond to \(P^G\) goals).

### 4.2. Introducing S-boxes

The “box” notion discussed previously is now formally defined in this section.

Let us consider the constraint system \(S = \{c_1, \ldots, c_m\}\) along with the Cartesian product of the variables’ domains occurring in some constraint of \(S\), \(D = D_1 \times \cdots \times D_n\). The declarative semantics \(S^*\) of \(S\) corresponds to the set of \(n\)-tuples of \(D\) satisfying the conjunction \(c_1 \land \cdots \land c_m\). In general, one must content oneself with an approximation \(\mathcal{S}\) of this semantics (e.g. with real-valued variables). This approximation is related to the greatest common fixed-point of the \(N[c_i]\) included in \(D\) [2]:

\[
\mathcal{S} = \max\{u \in \bigcap_{i=1}^m \text{fixed-pt}(N[c_i]) \mid u \subseteq D\}
\]

Let \(C = c_1 \land \cdots \land c_m\) be the constraint whose narrowing operator \(N[C]\) computes \(\mathcal{S}\). Then, the constraint set \(\{c_1, \ldots, c_m\}\) may be replaced by \(C\) without modifying the program semantics.

Constraint \(C\) may in the same way be integrated in another constraint \(C'\). A huge set of constraint may thus be organized into a hierarchy, preserving its semantics. This idea is at the root of the S-box notion:

**Definition 1 (S-box)** Let \(V\) be a set of \(n\) variables, \(D = D_1 \times \cdots \times D_n\) a Cartesian product of domains, and \(C\) a set of constraints. A S-box is a non-empty set \(\sigma = \{a_1, \ldots, a_m\}\), where \(a_i\) is either a constraint from \(C\) or a S-box. The narrowing operator \(N[\sigma]\) associated to the S-box \(\sigma\) is defined as:

\[
N[\sigma]: \quad D \rightarrow D
\]

\[
X \mapsto N[\sigma](X) = \max\{u \in \bigcap_{i=1}^m \text{fixed-pt}(N[a_i]) \mid u \subseteq X\}
\]

The operator associated to Algorithm NAR (Alg. 1) has the following properties [15]: it is contractant, correct, confluent, and monotone. Hence, it is a narrowing operator. Since we have also \(\text{NAR}(S, \{D_1 \times \cdots \times D_n\}) = \mathcal{S}\) for \(S = \{c_1, \ldots, c_m\}\), we deduce that NAR may be used in practice as an implementation of the narrowing operator \(N[\sigma]\) of an S-box \(\sigma\).

However, in order to fill the box notion given in the previous section, the narrowing operator \(N[\sigma]\) has to be atomically computed, that is, the domains’ propagation fixed-point must be reached in each S-box \(\sigma\) of a S-box hierarchy without reinvoking any constraint outside \(\sigma\). For example, consider the hierarchy corresponding to the store of Figure 3 (see the tree in Figure 6).

![Tree-structured representation of the S-box hierarchy in Figure 3](center_B)

**Figure 6. Tree-structured representation of the S-box hierarchy in Figure 3**

Operationally, assuming that each S-box owns its own propagation queue in which are pushed the constraints to
reinvoke belonging to that S-box, it is forbidden to go consider the propagation queue of one S-box “above” or at the same level as the current one if all the S-box queues “below” it (including its own) are not empty.

The difficulty arises from the fact that considering a constraint out of one queue may add a constraint to any queue in the tree. For example, assume that the current S-box is \( \odot \). Then, popping constraints out of its queue may add constraints into the queues of \( \odot \) and \( \odot \). It is then necessary to consider these queues before considering \( \odot \) and \( \odot \). However, processing \( \odot \) after having considered \( \odot \) may add new constraints to the queue of \( \odot \) such that it is necessary to return processing \( \odot \). This process will eventually terminate since all operators are contractant ones over a finite set of numbers. Nevertheless, we have to traverse the tree following a complicated path before reaching quiescence.

The idea is then to use only one propagation queue and to rely on the Dewey notation properties [12]. Given a S-box hierarchy represented as a tree, we associate to each constraint the Dewey index of its S-box in the tree-structured representation; During propagation, each constraint \( c' \) to be reconsidered is added in the propagation list at a position such that the list is sorted according to the order \( O_c \) defined below (with \( c \) being the last reinvoked constraint before adding \( c' \)).

**Notations.** Let \( P \) be a constraint program, and \( \sigma \) a S-box containing a constraint \( c \) of \( P \). The \( \mathcal{I}_c \) index of \( c \) corresponds to the Dewey index of \( \sigma \) in the S-box hierarchy of \( P \). Let \( \text{SzComPref}(\mathcal{I}_1, \mathcal{I}_2) \) be the function returning the size (number of signs) of the common prefix of indices \( \mathcal{I}_1 \) and \( \mathcal{I}_2 \).

**Definition 2 (Partial constraint ordering \( O_c \))** Let \( c_1 \), \( c_2 \), and \( c \) be three constraints. Constraint \( c_1 \) is said smaller than \( c_2 \) with respect to \( c \) \( c_1 \prec c_2 \) if and only if:

\[
\begin{align*}
   c_1 & \prec c_2 \iff \\
1. \ & \text{SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) > \text{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \\
2. \ & \text{or SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) = \text{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \quad \text{and} \quad \mathcal{I}_{c_2} >^{\text{lex}} \mathcal{I}_{c_1} \\
& \text{>(lex: lexicographical extension of > over naturals)}
\end{align*}
\]

Note that two constraints with the same index (that is, in the same S-box) are unordered with respect to any constraint.

**Example 2 (Partial ordering on constraints)** Let \( c_1 \), \( c_2 \), \( c_3 \), and \( c_4 \) be constraints with indices \( \mathcal{I}_{c_1} = 1.1.5, \mathcal{I}_{c_2} = 1.1.4, \mathcal{I}_{c_3} = 1.1.5.6, \) and \( \mathcal{I}_{c_4} = 3.1.4 \). We have the following inequalities:

\[
\begin{align*}
   & c_1 \prec c_2 \\
   & c_2 \prec c_1 \\
   & c_3 \prec c_4 \\
   & c_4 \prec c_3
\end{align*}
\]

The propagation strategy based on sorting the propagation queue with respect to the last popped constraint may then be proved to ensure atomicity when computing the fixed-point in the S-boxes by noting that Rule 1 ensures that constraints in the S-boxes closest to the S-box of \( c \) (last reinvoked constraint) are put at the head of the propagation queue (locality principle), and Rule 2 sorts constraints in the S-boxes “above” or at the same level than \( \sigma \) in such a way that the locality principle be respected for them when they are later considered.

Algorithm NAR needs to be slightly modified in order to take into account the new propagation scheme. The resulting algorithm called REVINCNAR (Alg. 2) is an incremental modified version of NAR supporting the S-box abstraction.

**Notations.** Let \( c_1, \ldots, c_m \) be the constraints appearing in the propagation list \( Q \) (in that order). Functions used in Algorithm 2 have the following meaning:

\[
\begin{align*}
\text{SBOX}(c) & : \text{returns the S-box containing } c; \\
\text{DEWEY}(c) & : \text{returns the Dewey index of Constraint } c; \\
\text{INSERT}(Q, c) & : \text{inserts the constraint } c \text{ in the propagation queue } Q \text{ just before the constraint } c_i \text{ such that:} \\
& \left\{ \begin{array}{lcl}
\forall j \in \{1, \ldots, i-1\} : \mathcal{I}_{c_j} \prec \mathcal{I}_c \\
\forall j \in \{i, \ldots, m\} : \mathcal{I}_{c_j} \succeq \mathcal{I}_c
\end{array} \right.
\end{align*}
\]

\( \text{where } \prec \text{ is the ordering on Dewey indices corresponding to the ordering } \prec \text{ on constraints, and } i \text{ is the Dewey index of the S-box containing the constraint being reinvoked;} \)

\[
\begin{align*}
\text{UNDERFOCUS}(\sigma) & : \text{returns true if the Dewey index of the S-box } \sigma_f \text{ having the focus is a prefix of the Dewey index of } \sigma \text{ (i.e. } \sigma \text{ is a “descendant” of } \sigma_f); \\
\text{TOP}(Q) & : \text{returns } c_i \text{ from } Q. \text{ Note: the constraint is not removed from } Q. \\
\text{POPHAND}(Q) & : \text{discards } c_1 \text{ from } Q.
\end{align*}
\]

**Note that Index } \iota \text{ has to be initialized to 1 (where “1” is the Dewey index of the S-box containing all the constraints of the debugged program) somewhere before the first call to REVINCNAR. Note also that } Q \text{ may be non empty when entering or leaving REVINCNAR. This is the case, for example, whenever the focus is not set to the whole store and there exists in } Q \text{ some constraints which are not under the focus.}
Algorithm 2. **RevIncNAR**: NAR revised to support S-boxes

RevIncNAR (in: \( N[c_1] \); inout: \( D = D_1 \times \cdots \times D_n \))

\[
\begin{align*}
Q &:= \text{insert}(Q, c_1) \quad \% \text{Cstr added to propag. list} \\
S &:= S \cup \{c_1\} \quad \% \text{Constraint added to the store}
\end{align*}
\]

\[
\text{while } Q \neq \emptyset \text{ and } \text{underFocus(SBox(Top(Q)))} \quad \text{and } D \neq \emptyset \text{ do}
\begin{align*}
(Q, c) &:= \text{popHead}(Q) \\
\epsilon &:= \text{dewey}(c) \\
D' &:= N[c](D) \\
\text{if } (D' \neq D) \text{ then}
\begin{align*}
\text{forall } \{c_j \in S | \exists x_k \in \text{Var}(c_j) \land D_k' \neq D_k \} &\text{ do}
Q &:= \text{insert}(Q, c_j)
\end{align*}
\end{align*}
\]

\[
D := D'
\]

\end{while}
\]

5. Presentation of the debugger prototype

A prototype for a S-box based debugger has been implemented over clp(fd) [6], an extension of Prolog by Codognet and Diaz for constraint logic programming with integer-valued variables. Functionalities of the prototype are briefly surveyed here. The reader is referred to [3] for an in-depth description of the debugger implementation.

Figure 7 displays a snapshot of the debugger execution on Cars, a sequencing problem extracted from the examples of the clp(fd) distribution.

Figure 7. Debugging cars.pl without S-boxes

The main window is composed of two parts:
1. an editor containing the source code to interpret;
2. a console for the clp(fd) system where goals may be written interactively.

S-box creation may be done in the editor by tagging with the mouse some sets of constraints and/or goals, and giving them a name before running the interpreter.

The debugger permits visualizing variable domain modifications graphically. Constraints and variables appear in the S-boxes with the same aspect as in the source code (e.g. variable names are preserved, contrariwise to the habit in Prolog debuggers).

A window is associated to each S-box (cf. Fig. 8). Any of them may be masked or displayed at the user will. A S-box appears like an ordinary constraint in the window of its ancestor.

Breakpoints may be added to variables (break when a domain is equal/different to another, reduced to one value, etc.), constraints (break on re-invocation), S-boxes (break on entering/leaving), and other events such as failure or entering in the labeling phase. The propagation process may be traced step-by-step, allowing the programmer to understand the relations between sets of constraints/S-boxes.

6. Conclusion

We have presented the S-box abstraction that permits introducing in the store the semantics of sets of constraints which is present in the program. S-boxes allow us to design a new kind of debugger that focuses on the constraint store and the propagation process of variables’ domains narrowing. By using S-boxes, the store is no longer a huge and flat collection of constraints, but a tree structure with only few “user global constraints” the programmer may inspect in a handy way by zooming in and out the S-boxes he wants to consider. The inspection of the store is made easier by the fact that these global constraints have the semantics the programmer gave in first place to the corresponding sets of constraints in its source code.

The S-box abstraction required the modification of the propagation algorithm used in constraint solvers. We have given an incremental version of such an algorithm which is only a slight modification of the traditional algorithm, thus permitting to replace it smoothly by our version supporting S-boxes. A prototype of a debugger with S-boxes has been implemented in order to validate our ideas. It appeared that the required modifications of the host solver were few, which is encouraging for future portability on other solvers.

The method used to support S-boxes is mainly based on modifying the propagation order for re-invocation of constraints. Wallace and Freuder [19] have shown that considering in a precise order constraints to be reinvoked could
speed-up the quiescence of the constraint network, and then the solving process. For example, this is the case when first considering constraints containing variables occurring many times in the store. Such a strategy could be modeled by S-boxes. Extending the S-box abstraction to permit the use of optimizing heuristics is a direction for further research.

Acknowledgements

The research exposed here was supported by the LTR DiSCIPl European ESPRIT Project #22532. Discussions with the researchers involved in this project are gratefully acknowledged.

References