Debugging Constraint Programs by Store Inspection

Frédéric Benhamou and Frédéric Goualard
The DiSCiPl Consortium

October 12, 1998

Abstract

In this paper, the debugging of constraint programs is discussed: two separate debugging levels are isolated, one of them —namely the *store level*— being shown to be unreachable from "traditional debuggers". A method permitting to debug programs by inspecting the store and the domain variations of the variables is described. The necessary new algorithms along with the modifications for the existing ones are given. Finally, an implementation in the Constraint Logic Programming framework is studied.

1 Introduction

Constraint Programming [8] (CP) is a language-independent paradigm where a problem is solved by specifying the properties its solutions must verify. Resolving problems that way leads to programs which are both elegant and efficient (in some cases, Constraint Programming is known to outperform state-of-the-art specialized algorithms).

Since its introduction in the drawing tool SKETCHPAD [24], Constraint Programming has raised up a growing interest among researchers such that many languages have been extended to support it (*e.g.* REF-ARF [15], for procedural languages; CHIP [1], Prolog IV [6], clp(BNR) [5], clp(fd) [11], for logic languages; ThingLab [7], ILOG Solver [22], for object-oriented languages; HOPE [13] for functional languages).

Nevertheless, Constraint Programming is not as widely used as it could be by industrials. This seems to be due from the lack of programming environments offering advanced features to the user, such as help for debugging and profiling its programs (if we except the tools described in [18] and [23]). The expressiveness of Constraint Programming allows shortening programs at the cost of forbidding the use of traditional debuggers such as step-by-step tracers, since most of the work is hidden behind constraint processing (see example 1 page 4).

In this paper, we describe a new kind of CP debugger which reveals to the user what is usually done behind his back, by displaying constraint processing in a usable way. The paper is organized as follows: section 2 briefly introduces some fundamental Constraint Programming notions; section 3 describes the difficulties the user has to face when debugging constraint programs with traditional tools; section 4 presents a technique allowing to debug constraint programs at a different level than the one covered by traditional debuggers; the algorithms supporting it are described along with an example of use; section 5 presents the implementation and the functionalities of a prototype instanciating our ideas in the Constraint Logic Programming framework; finally, we conclude by pointing out the advantages and weaknesses of the new debugging system.

2 Preliminary notions

For the sake of simplicity, we will restrict ourselves in the following to *Constraint Logic Programming* (CLP) —that is, Constraint Programming embedded in Logic languages (mainly Prolog [12])— and to numerical constraints involving only integer-valued variables.

2.1 Handling constraints using local consistencies and filtering

Consider the constraint

$$c_1 : x + y = z \quad \text{where} \quad \begin{cases} x \in D_x = \{2, 3, \ldots, 6\}, \\ y \in D_y = \{2, \ldots, 4\}, \\ z \in D_z = \{3, \ldots, 5\} \end{cases}$$

One may note that some values in the domains of the variables are clearly inconsistent with c_1. For example, x cannot take the value 6 as there is no value in the domain of y such that $x + y$ would be a value of the domain of z. A function discarding all the values of a Cartesian product of domains for which a constraint c is inconsistent will be called a *Constraint Narrowing Operator* N_c for c (referred as a CNO in the following). This is related to *local consistency techniques* such as *arc-consistency* [17, 19].

Applying N_{c_1} to $D_x \times D_y \times D_z$ would narrow them down to $D'_x = \{2, 3\}$, $D'_y = \{2, 3\}$, and $D'_z = \{4, 5\}$. The values of these domains are the only ones such that, given one of the variables and its associated domain D, there exists for every value of D a possible assignment for all the remaining variables such that the constraint holds.

The description of the algorithms used to implement this kind of contracting operators is beyond the scope of this paper. The interested reader is referred to [10, 11] among other references.

If x is later known to be different from 2 (*e.g.* by way of the addition of the constraint $c_2 : x > 2$), N_{c_1} will be able to narrow down further D'_x, D'_y, and D'_z to $D''_x = \{3\}$, $D''_y = \{2\}$, and $D''_z = \{5\}$.

More generally, the constraints appearing in a problem form the nodes of a network (called a *constraint store*) where two constraints are linked whenever they share at least one variable (see an example in figure 1).

The constraint system is then solved by propagating the domain modifications induced by the applications of constraint narrowing operators to all the constraints involving the modified variables through the links of the constraint network until reaching a stable state. The algorithm 1 describes the introduction of a constraint into the store: the constraint is added to the store and

$$\begin{cases} x^2 = y \\ x + y = z \\ z = k + y \\ t = k - 5 \end{cases}$$

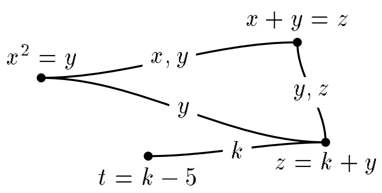

Figure 1: A constraint system and its associated *store*

to a *propagation queue* (managed as a FIFO list in most of the implementations); its narrowing operator is applied to the domains of variables, and all the constraints involving modified variables are pushed into the propagation queue. The process terminates on failure (*i.e.* a variable domain is empty) or when the propagation queue is empty (the fixed-point of all the contracting operators is reached). The consistencies used are only partial consistencies. As a consequence, domain propagation alone is, in general, incomplete. Then, a *labeling phase* —consisting in assigning to each variable all of its possible remaining values through backtracking— is done when all constraints have been added to the store. It allows to find all the consistent assignments of the variables involved in the constraint system.

Algorithm 1: The incremental Narrowing algorithm

IncNar(**in** (c_1, N_1) ; **inout** $D = D_1 \times \cdots \times D_n$)
begin
    $\mathcal{Q}$:= $\{c_1\}$ % Constraint added to the propagation queue
    $\mathcal{S}$:= $\mathcal{S} \cup \{c_1\}$ % Constraint added to the store
    while $\mathcal{Q} \neq \emptyset$ **and** $D \neq \emptyset$ **do**
        c := choose one c_i in $\mathcal{Q}$
        D' := $N_{c_i}(D)$
        if $D' \neq D$ **then**
            $\mathcal{Q}$:= $\mathcal{Q} \cup \{c_j \in \mathcal{S} \mid \exists x_k \in \mathsf{Var}(c_j) \wedge D'_k \neq D_k\}$
            D := D'
        endif
        $\mathcal{Q}$:= $\mathcal{Q} \setminus \{c\}$
    endwhile
end.

3 Difficulties arising when debugging constraint programs

In this section, the difficulties encountered when one wants to debug a constraint based program are described.

1

3.1 The Constraint Programming efficiency

Consider the example given below of a "problem" which benefits from the use of Constraint Programming:

Example 1 (The Yoshigahara problem). Given $A, \ldots, I$, nine integer valued variables, use all digits 1 through 9 only once such that

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \tag{1}$$

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

The Yoshigahara problem has only one solution (up to permutations of the quotients): $\frac{5}{34} + \frac{7}{68} + \frac{9}{12} = 1$

A naive resolution of this problem involves the generation of the 9^9 possible assignments to variables A through I, followed by the test whether each assignment meets the relation (1). This is the so-called *generate and test* approach. It is illustrated by the Java program given in program 1.

On the other hand, a constraint-based resolution starts stating all the constraints before generating possible assignments. As a result, many branches of the assignment tree are discarded early, allowing to remove at once a lot of inconsistent assignments. This is the *test and generate* approach illustrated by the DecLIC [4] program given in program 2, page 5.

On an AMD K6-166MHz based PC, all solutions are found by the program 1 in 1778.2s. and by the program 2 in 1.83s, which shows the superiority of Constraint Programming for such kind of problems.

3.2 Drawbacks of the expressiveness of CP

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 true 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. 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. A specialized tool able to display the constraint store in a readable way must be devised for this purpose.

Program 1: The *Yoshigahara* problem solved by a naive Java program

```
class yoshigahara {
  private static boolean alldifferent(int a, int b, int c, int d, int e,
                                      int f, int g, int h, int i)
    {
      return (a!=b && a!=c && a!=d && a!=e && a!=f && a!=g && a!=h && a!=i &&
              b!=c && b!=d && b!=e && b!=f && b!=g && b!=h && b!=i && c!=d &&
              c!=e && c!=f && c!=g && c!=h && c!=i && d!=e && d!=f && d!=g &&
              d!=h && d!=i && e!=f && e!=g && e!=h && e!=i && f!=g && f!=h &&
              f!=i && g!=h && g!=i && g!=i);
    }
  public static void main(String[] args)
    {
      for(int A=1;A<=9;A++)                                          // Generate
        for(int B=1;B<=9;B++) for(int C=1;C<=9;C++)
            for(int D=1;D<=9;D++) for(int E=1;E<=9;E++)
                for(int F=1;F<=9;F++) for(int G=1;G<=9;G++)
                    for(int H=1;H<=9;H++)
                      for(int I=1;I<=9;I++) {
                        if (alldifferent(A,B,C,D,E,F,G,H,I)) {           // Test
                          float res=(float)A/(10*B+C)+(float)D/(10*E+F)+
                                    (float)G/(10*H+I);
                          if (res == 1.0)
                            System.out.println(A+"/"+B+C+" + "+D+"/"+E+F+
                                               " + "+G+"/"+H+I+" = 1");
                        }
                      }
    }
}
```

Program 2: the *Yoshigahara* problem solved by a DecLIC program

```
:- L=[A,B,C,D,E,F,G,H,I],
   is_integer(L),                                                    % Test
   alldifferent(L),
   domain(L,1,9),
   A/(10*B+C)+D/(10*E+F)+G/(10*H+I) $= 1,
   labelingff(L).          % Generate (variable with least possible values labeled first)
```

4 Focusing on the store

Constraint debugging tools such as the ones described in [18] and [23] focus on the displaying of the variable domains in “real time.” This is an important functionnality. However, we believe that it is insufficient to fully debug constraint programs, particularly when dealing with real-valued variables whose 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 (see figure 14, page 19). Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (*e.g.* in Prolog IV), and constraints stored are not always those given by the user (*e.g.* some systems such as clp(BNR) decompose constraints into some primitives, using new variables).

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

4.1 Structuring the store

At this point, the main idea is to use the following properties of IncNar [20]:

> IncNar *is correct, confluent, contractant, and monotone.*

These are exactly the properties defining a Constraint Narrowing Operator [3]. Thus, given a set of m constraints involving n variables, with their associated CNOs

$$S = \{(C_1, N_1), \ldots, (C_m, N_m)\}$$

and the Cartesian product of domains $D = D_1 \times \cdots \times D_n$, we may replace S by the pair (C', N') where:

$$\begin{cases} C' = & C_1 \wedge \ldots \wedge C_m \\ N' = & \textsc{IncNar}(\{(C_1, N_1), D) \text{ followed by} \\ & \textsc{IncNar}(\{(C_2, N_2), D^1) \text{ followed by} \\ & \ldots \text{ followed by} \\ & \textsc{IncNar}(\{(C_m, N_m), D^{m-1}) \end{cases} \tag{2}$$

where D^i is the contracted Cartesian product of domains resulting from the application of IncNar to $C_1, \ldots, C_i$.

More generally, thanks to the confluency of IncNar, it is possible to gather arbitrarily some constraints without modifying the declarative semantics of the original constraint system. This implies that it is possible to structure hierarchically the global store (containing all constraints) into a conjunction of sub-stores gathering only some of the constraints. Each sub-store may be viewed as a new constraint defined by its input and output (Cartesian product of the domains of the variables involved). The idea is then to let the user focus on the sub-store he is interested in, throwing temporarily away all the constraints not included in it.

A pair (C', N') of the form (2) will be called a *S-box* where S stands for *Store.*

4.2 Using S-boxes

We describe in the following how S-boxes are created by the user and what can be done with them with respect to debugging.

The creation of a S-box is done by tagging some goals in the source code. All the constraints added to the store while executing these goals are put in the same S-box. For example, consider the CLP program P (program 3 page 7) aimed at finding the time T at which a ball moving along a curve will hit a wall [9].

Program 3: The program P

```
object_A(X,Y,Z):-                                   % Shape of the wall
        X ≤ 0, Y ≤ 0, Z ≤ 0.
object_B(XCx,YCy,ZCz):-                             % Shape of the ball
        XCx^2 + YCy^2 + ZCz^2 = 1.
center_B(T,Cx,Cy,Cz):-             % Pos. of the center of the ball at T
        T^2 - Cx = 10,
        2T - Cy = 10,
        Cz - T^2 + 7T = 10.
object_B_moving(T,X,Y,Z):-         % Pos. of point (X,Y,Z) in ball at T
        center_B(T,Cx,Cy,Cz),
        XCx = X - Cx,
        YCy = Y - Cy,
        ZCz = Z - Cz,
        object_B(XCx,YCy,ZCz).
:- T ≥ 0,
   object_A(X,Y,Z),
   object_B_moving(T,X,Y,Z).
```

A natural use of the S-boxes is to tag all the goals of P, giving us the store structure described in figure 2. It is then possible to view each S-box as a single constraint involving all the variables appearing in the constraints it embedds. This simplifies greatly the store by masking many constraints while preserving the declarative semantics. For example, the store of figure 2 is best viewed as the simplified store of figure 3. Note that the number of links between each "constraint" has dramatically decreased. It is thus simpler to add some "guards" to these links warning the user of some violation of a property when propagating domains from a S-box to another. For example, assume that the propagation of some domains given by the user as an input to the box $C_2(T, X, Y, Z)$ does no lead to the right domains as output; in order to find what went wrong, the user can focus on the S-box of $C_2(T, X, Y, Z)$, obtaining the store described in figure 4.

The action of *focusing on a S-box* σ means restricting the set of constraints present in the store to the ones contained directly or indirectly in σ. As a consequence, the *focus* must be reset to the entire store sooner or later during the debugging process, such that all the constraints have a chance to be reconsidered.

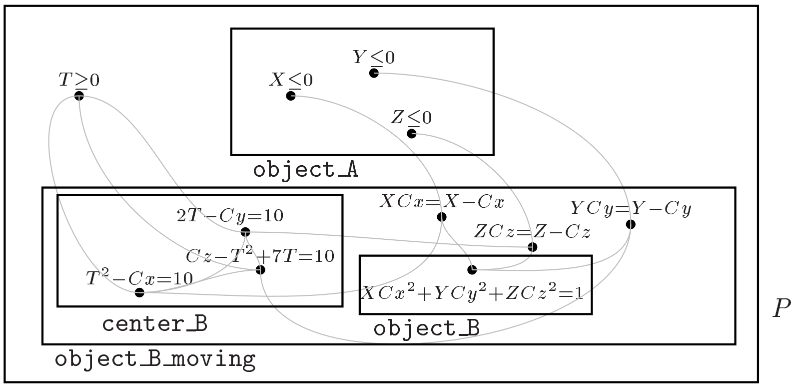

Figure 2: Store structure for program P when tagging all goals

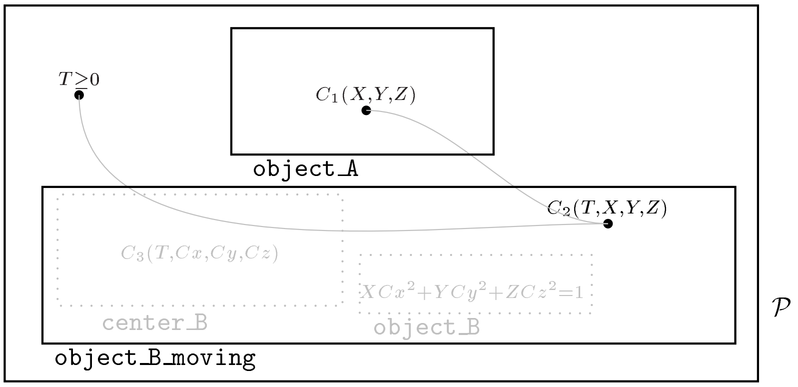

Figure 3: Abstraction for the store of the program $\mathcal{P}$

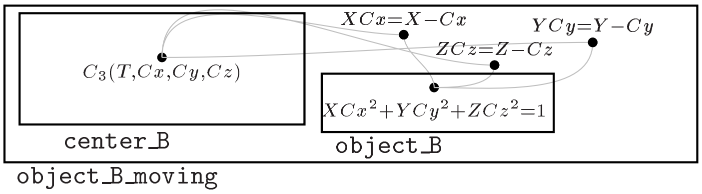

Figure 4: The new store after focusing on $C_2(X, Y, Z)$

4.3 Practical issues

In order to fulfil the S-boxes definition, it is necessary to reach the fixed-point of domain propagation in each S-box *in turn*. The constraints enforced on the propagation order are best viewed by reasoning on the tree representation of the S-box hierarchy. For example, the hierarchy given in figure 2 is equivalent to the tree drawn in figure 5.

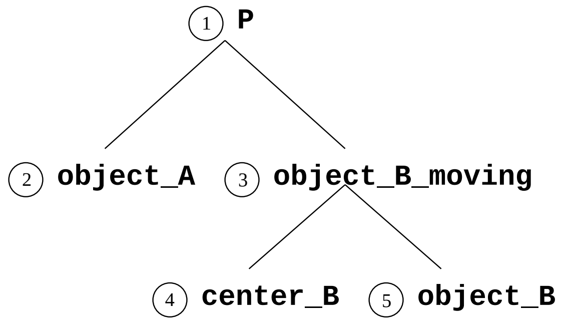

Figure 5: Tree representation of the S-box hierarchy of figure 2

Assuming that each S-box owns its own propagation queue in which are put the constraints to re-invoke belonging to that S-box, it is forbidden to go relax the propagation queue of one S-box “above” or at the same level as the current one if all the S-boxes queues “below” it (including its own) are not empty. The difficulty arises from the fact that from one queue, a constraint may be added to any queue in the tree. For example, assume that the current S-box is ③. Then, relaxing its queue may add constraints into the queues of ④ and ⑤. It is then necessary to relax these queues before considering ① and ②. However, relaxing ⑤ after having relaxed ④ may add new constraints to the queue of ④ such that it is necessary to return relaxing ④. Of course, everything becomes worst with deeper trees. Moreover, the resulting propagation algorithm takes an exponential time to relax all the S-boxes.

The idea is then to use only one propagation queue as it was done when no S-boxes were involved, and to rely on the properties of the Dewey notation [16]. Given a S-box hierarchy represented as a tree, we associate to each node (and then, to each corresponding S-box) its index in the Dewey notation (see figure 6).

To each constraint is associated at runtime the Dewey index of the S-box containing it. During the relaxation process, each constraint is added to the propagation queue at a position such that the queue is always sorted according to the ordering $\mathcal{O}$ defined as follows:

Definition 1 (Ordering $\mathcal{O}$ of constraints) *Let $\mathcal{I}_c$ be the Dewey index of the S-box to which c belongs. Given c_1, c_2 and c_3 three constraints, the constraint c_1 is said to be* smaller than c_2 wrt. c_3 *($c_1 \overset{c_3}{\prec} c_2$) iff:*

1. *$\mathcal{I}_{c_1}$ matches lexicographically $\mathcal{I}_{c_3}$ more than $\mathcal{I}_{c_2}$, or*
2. *$\mathcal{I}_{c_1}$ and $\mathcal{I}_{c_2}$ match equally $\mathcal{I}_{c_3}$ and $\mathcal{I}_{c_1}$ is “before” $\mathcal{I}_{c_2}$ in the lexicographic order.*

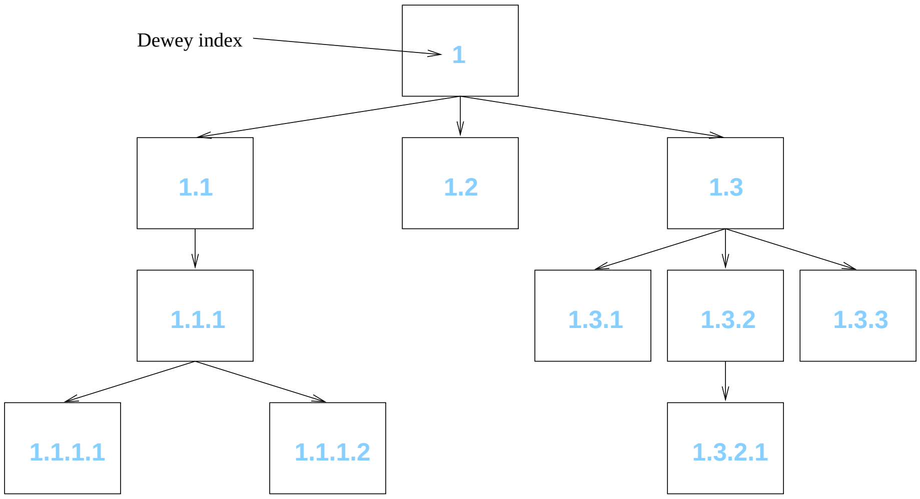

Figure 6: Hierarchy of S-boxes with the Dewey notation

Alternatively, the same ordering may be defined for Dewey indices directly or for S-boxes:

$$\mathcal{I}_{\sigma_1} \overset{\mathcal{I}_{\sigma_3}}{\prec} \mathcal{I}_{\sigma_2} \Longleftrightarrow \sigma_1 \overset{\sigma_3}{\prec} \sigma_2$$

where σ_i is the S-box containing c_i and $\mathcal{I}_{\sigma_i}$ is the Dewey index of the S-box σ_i.

One may easily convince oneself that the propagation order obtained by always putting a constraint in the queue at a position such that the order is preserved wrt. *the last popped constraint*, meets the requirements given above for a propagation strategy to suppport S-boxes.

The algorithm IncNar given page 3 needs to be slightly modified in order to take into account the new propagation scheme. The resulting algorithm, called RevIncNar (*Revised Incremental Algorithm*) is described by the algorithm 2.

Let $c_1, \ldots, c_m$ be the constraints appearing in $\mathcal{Q}$ (in that order). The functions introduced in algorithm 2 have the following meaning:

SBox(c) : returns the S-box containing c;

Dewey(σ) : returns the Dewey index of the S-box σ;

Insert$_{\mathcal{O},\iota}(\mathcal{Q}, c)$: inserts the constraint c in the propagation queue $\mathcal{Q}$ just before the constraint c_i such that:

$$\begin{cases} \forall j \in \{1, \ldots, i-1\} : \mathcal{I}_{c_j} \overset{\iota}{\prec} \mathcal{I}_c \\ \forall j \in \{i, \ldots, m\} : \mathcal{I}_{c_j} \overset{\iota}{\succeq} \mathcal{I}_c \end{cases}$$

UnderFocus(σ) : returns true if the Dewey index of the S-box σ_f having the focus is a prefix of the Dewey index of σ (*i.e.* σ is a descendant of σ_f);

Top($\mathcal{Q}$) : returns c_1 from $\mathcal{Q}$. Note: the constraint is not removed from $\mathcal{Q}$;

Algorithm 2: RevIncNar: IncNar revisited to support S-boxes

```
RevIncNar(in (c_1, N_1) ; inout D = D_1 × ··· × D_n)
begin
    Q := Insert_{O,ι}(Q, c_1)  % Constraint added to the propagation queue
    S := S ∪ {c_1}  % Constraint added to the store
    while not(Empty(Q)) and UnderFocus(SBox(Top(Q))) and D ≠ ∅ do
        (Q,c) := PopHead(Q)  % Cstr. at top of Q is assigned to c, then removed
        ι := Dewey(SBox(c))
        D' := N_c(D)
        if D' ≠ D then
            forall {c_j ∈ S | ∃x_k ∈ Var(c_j) ∧ D'_k ≠ D_k} do
                Q := Insert_{O,ι}(Q, c_j)
            endforall
            D := D'
        endif
    endwhile
end
```

PopHead($\mathcal{Q}$) : discards c_1 from $\mathcal{Q}$.

Note that ι must be initialized somewhere outside of RevIncNar before the first call to the algorithm. An obvious initialization is ι := 1 (where "1" is the Dewey index of the whole store). Note also that $\mathcal{Q}$ 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 $\mathcal{Q}$ some constraints which are not under the focus.

5 A S-boxes based debugger

A debugger based on the S-box notion presented previously has been implemented on top of clp(fd) [11], an extension of Prolog supporting constraint programming over Finite Domains (positive integer-valued variables), designed by Philippe Codognet and Daniel Diaz. The implementation along with the functionalities of this tool are surveyed in the following two sections.

5.1 Implementation

Though efficiency is not the primary concern for a debugger, a quick handling of the user interactions is an important factor for the ease of use of the graphical user interface. This point is in favor of a tight integration of the debugger into the solver such as to minimize the overhead induced by the necessary communications between them.

On the other hand, a fully integrated debugger cannot be used with another solver without rewriting it entirely.

In order to preserve portability while avoiding the overhead of the communications between a debugger completely separated from the solver, our debugger has been devised in two parts (see figure 7):

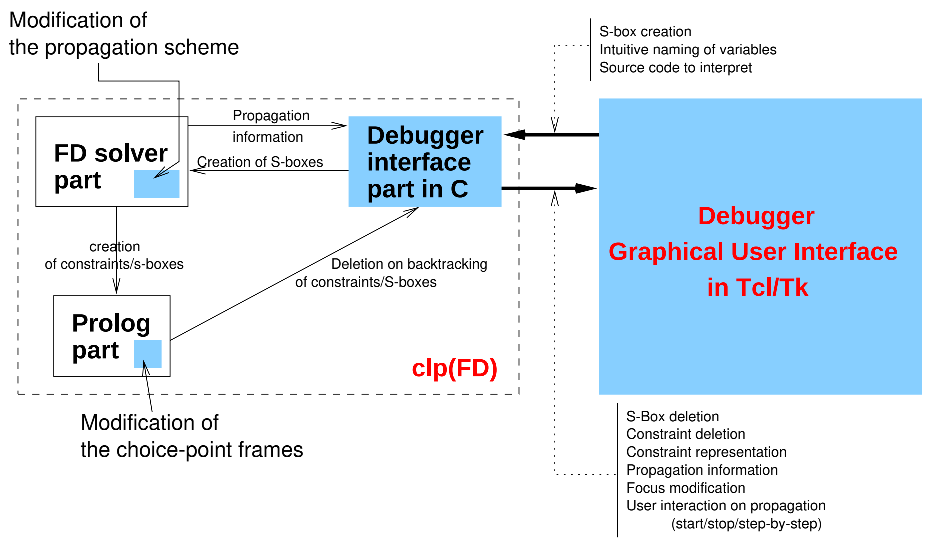

Figure 7: The debugger architecture

1. a tightly integrated part written in C interacting at a a low level with the solver; it implements some primitives such as methods specifying how a constraint must be displayed, and some structures entirely devoted to debugging (*e.g.* everything related to the S-boxes data);

2. a part written in Tcl/Tk [21] implementing all the graphical user interface. This part has its own representation of the structures related to the store and the S-boxes in order to minimize as much as possible the (costly) message passing between C and Tcl/Tk.

Filled frames in figure 7 represent the parts which have been added or modified in clp(fd). As one may see, the separation between the solver and the debugger is not as clean as what one would like it to be:

- the choice-point frame structures of the Prolog engine has been modified in order to keep trace of the contraints and S-boxes added to the store. This permits to erase them from the graphical representation in the GUI on backtracking. Figure 8 presents the resulting modified choice-point frame;

- the clp(fd) propagation process has been replaced by the one described above.

Actually, it would be very hard to devise a S-boxes based debugger without achieving these modifications of the kernel. The only alternative would be to integrate the debugger in a system providing the necessary handlers to obtain the same functionalities.[1]

Table 1 gathers all the registers added to the ones already present in clp(fd). Their use is presented in the following sections.

[1]Though such a modularity is difficult to achieve in a monolithic system such as a Prolog engine, it is quite natural for an object-oriented solver such as, say, Ilog Solver [22].

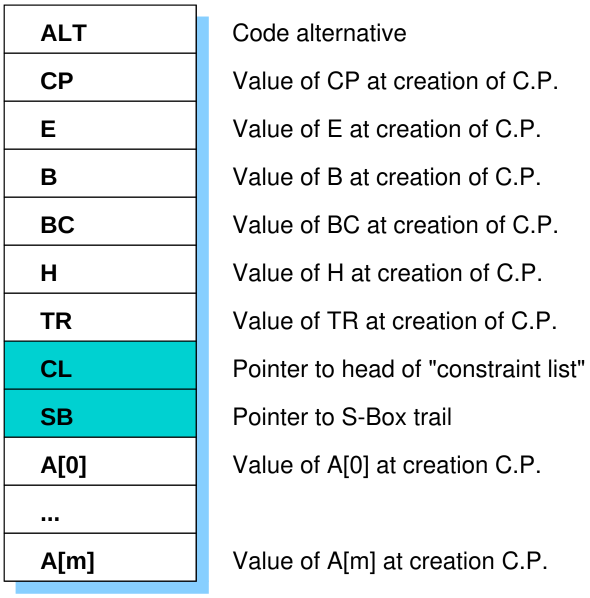

Figure 8: Modified choice-point frame

Table 1: Registers added to clp(fd)

name	**meaning**	**description**
SBF	*S-Box Focus*	pointer to the S-box having the focus
SBC	*S-Box Current*	pointeur to the current S-box *i.e.* the last one created but not finished
PLH	*Propagation List Head*	—

5.1.1 Modification of the backtracking process

A trailing structure (see figure 9) is created each time the Prolog engine creates a new choice-point. Actually, the information related to a S-box cannot be put in the Prolog trail since the S-boxes do not follow the clausal structure of the program (a S-box may be created in a clause and closed in another —see program 4).

Program 4: S-boxes creation *vs.* clausal structure

```
p:- begin_sbox, q, r.                 % S-box 1.1 created
q:- t, end_sbox, s.                   % S-box 1.1 closed
t:- begin_sbox, s, end_sbox.          % S-box 1.1.1 created and closed
```

As a consequence, all the S-boxes-related structures needing to be reset on backtracking are saved in a local specialized trail whose address is kept in the choice-point frame.

5.1.2 Modification of the propagation process

In clp(fd) [14], the propagation of domain modification is done as follows: each time the domain of a variable v is modified, the variable frame of v is put at the end of a propagation queue (handled as a FIFO list). Relaxation is achieved by popping the variable frame at the head of the queue and by reinvoking all the constraints whose addresses are kept in the frame (these are the constraints involving an occurrence of the variable). These reinvocations may add new variable frames at the end of the queue. Propagation ends when the queue is empty. Note that, here, the propagation queue is "*variable-oriented*" since it is a queue of variables.

On the other side, the algorithm 2 presented above uses a "*constraint-oriented*" view where the propagation queue contains only constraints. As a consequence, all the propagation process needs to be modified in order to support the S-boxes. Four fields have then been added to the original constraint frame (see the resulting frame in figure 10): the propagation queue is implemented by linking together the constraint frames of the constraints to reinvoke by using the field `Next_In_Queue`. The head of the propagation queue is always pointed by the register PLH. Note that we only need to know the position of the head since the introduction of a constraint in the queue is done by finding a position preserving the order $\mathcal{O}$ starting from the head. The field `Already_In_Queue` prevents us from pushing a constraint in the queue more than once.

The register SBC contains the name of the current S-box. This permits to know easily to which S-box a newly created constraint belongs to: at the creation of a constraint frame, the contents of SBC is copied into the field `Sbox_Ptr`.

The last field, `Cstr_Id`, is used as an identifier for the constraint when communicating with the Tcl/Tk interface. It permits, *e.g.* , to ask to the interface to highlight the graphical representation of the constraint each time it is reinvoked by the solver.

The original variable frame has been but slightly modified since the only modifications lie in the removing of the unused fields, namely all the ones which

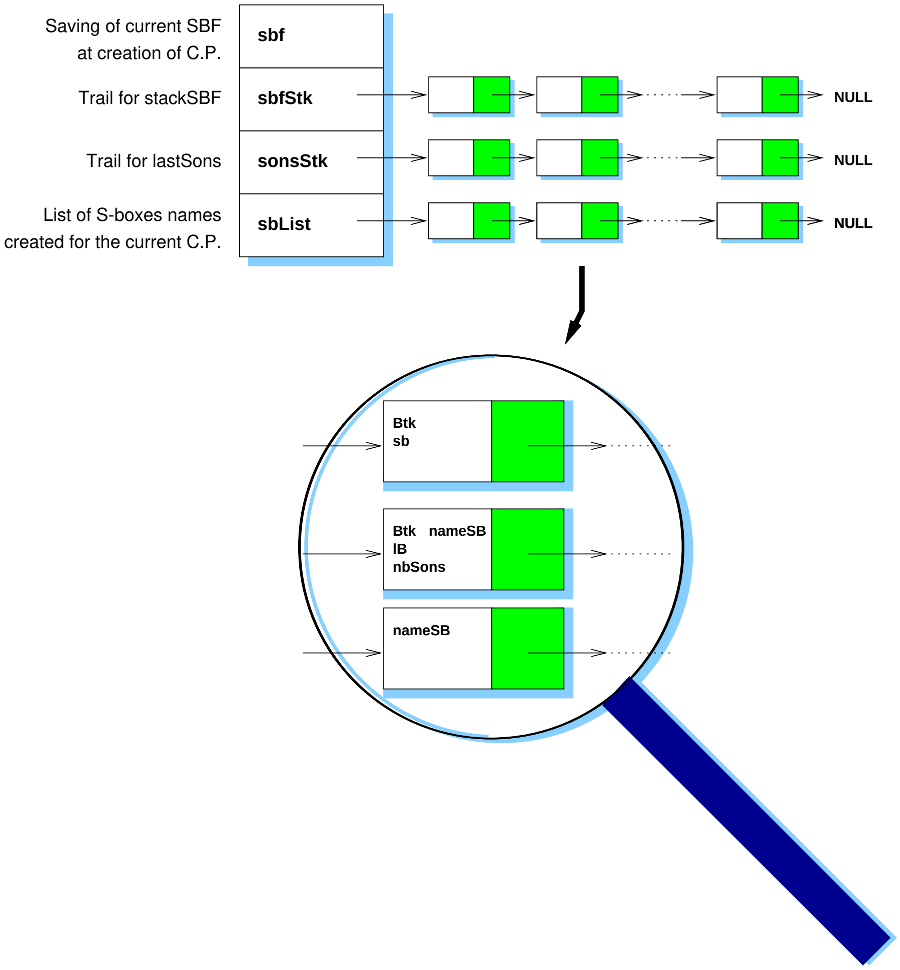

Figure 9: S-box trail frame

Next_In_Queue	Pointer to the next cstr. frame in propag. queue
Already_In_Queue	Is the cstr. already in propag queue?
Sbox_Ptr	Id. of the S-box containing the cstr.
Cstr_Id	GUI Id. of the constraint
Cstr_Address	Pointer to C function coding the constraint
Tell_Fdv_Adr	Address of constrained C-variable
AF_Pointer	Pointer to constraint environment

Figure 10: Modified constraint frame

where used to link the frames into a propagation queue. The resulting frame is depicted in picture 11.

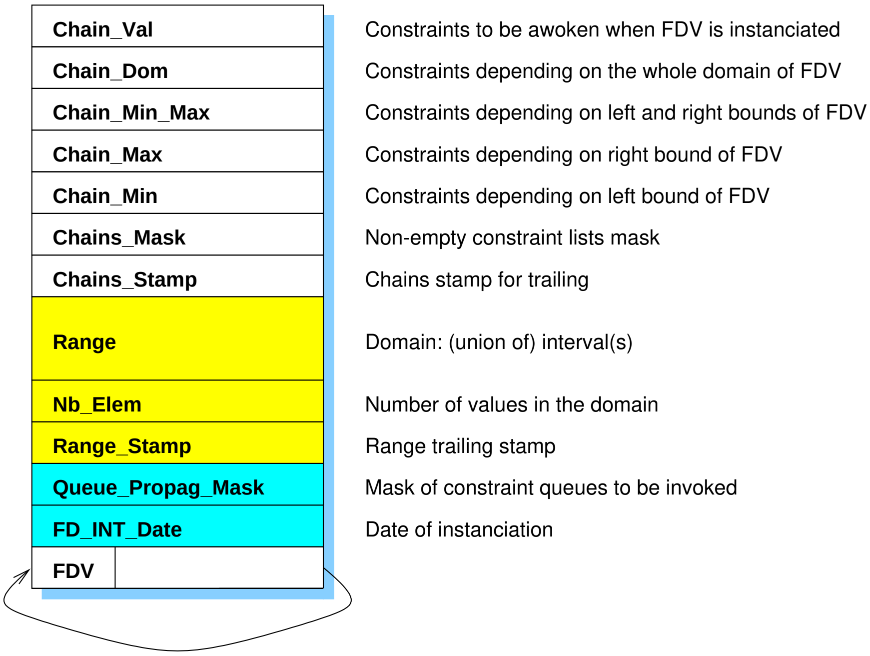

Figure 11: Modified variable Frame

5.1.3 Handling the S-boxes

S-boxes are created by tagging with the mouse some goals in the debugger editor (see section 5.2). Before being interpreted, the code contained in the editor is merged with those tags in a new CLP program in the following way: for each tag on a set of goals $g_1, \ldots, g_m$, the instructions `begin_sbox`[2] and `end_sbox` are added, respectively, before g_1 and after g_m. They are in charge of creating and "closing" at runtime the corresponding S-box.

Example 2 (Translation of a tagged program).

```
p:- a, b, c, d.  |  p:- a, begin_sbox, b, c, end_sbox, d.
```

Each S-box is uniquely identified by its Dewey index (see section 4.3). A natural representation of this identifier is a character string. Thus, the "S-boxes names" manipulated have the form,*e.g.* `"1.2.3"`. An advantage of such a representation lies in that the implementation of the ordering $\mathcal{O}$ becomes a straightforward comparison of strings.

The identifier of a S-box to be created is determined as follows: a "last sons" stack (see figure 12) contains the number of sons of each S-box along with some

[2]The instruction `begin_sbox` accepts an argument which is the symbolic name of the S-box chosen by the user. It is omitted here for the sake of clarity.

additional information used for backtracking purposes; the name of a new S-box is then the concatenation of the contents of the field `nameSB` on top of the stack (the current S-box) with the number of sons incremented by one.

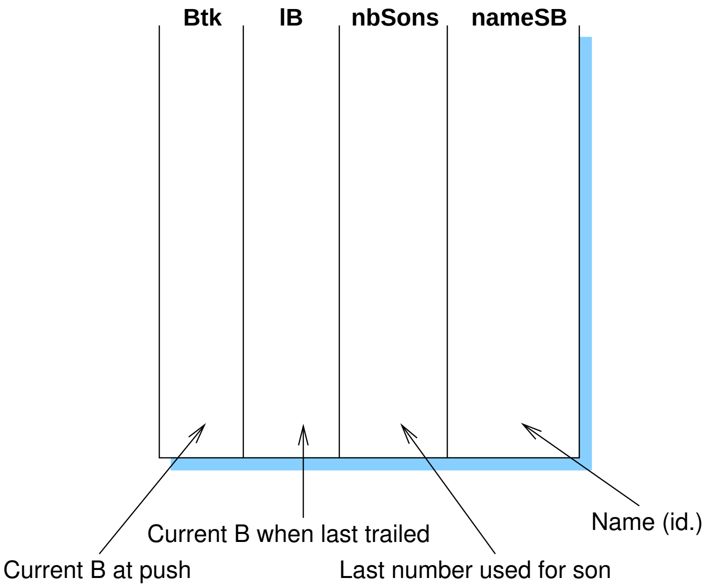

Figure 12: “Last sons” stack

On creation of a new S-box σ, the focus is changed to σ since the propagation cannot be done outside of σ until it is closed (otherwise, σ could not be viewed as a constraint like the primitive ones). It is then necessary to keep a stack of the names of the S-boxes having the focus in order to be able to reset the focus to its previous value when a S-box is closed. This is done by the SBF-*stack* described in figure 13.

Note that the identifiers of the S-boxes are not duplicated in the different structures (SBF stack, last sons stack,...) since they hold only pointers to them. As a consequence, it is easy to free the memory space used to represent these strings on backtracking, using the field `sb List` of the S-box trail.

5.2 Presentation of the tool

A quick overview of the debugger is presented in the following. The reader is referred to the soon-to-come manual [2] for an in-depth description of all its functionalities.

Figure 14 presents a snapshot of the execution of the debugger on `cars`, a sequencing problem extracted from the examples of the clp(fd) distribution. 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.

Once launched (by pushing one of the buttons of the toolbar), the interpretation of the code contained in the editor is done by clp(fd). It is possible to use

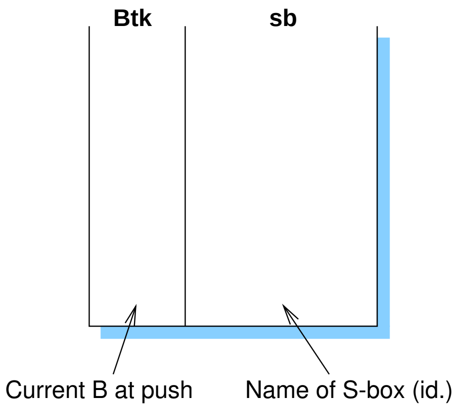

Figure 13: SBF stack

the embedded Prolog debugger to run the program step-by-step. The domain propagation may be stopped and restarted at will by the user using the buttons on the toolbar. The store is displayed in a separate window (visible on the foreground of the snapshot) in the form of a constraint network, as discussed previously. Additionnally, there are two windows dedicated to the displaying of the variable domains (graphically or textually). They are visible in figure 15.

Picture 14 presents a debugging session where S-boxes are not in use: there is only one store window displaying an unmanageable constraint network. Picture 15 presents the same debugging session where some goals of the program have been tagged to create S-boxes. There is one window for each S-box (each window may be individually withdrawn at will).

5.2.1 Breakpoints

The debugger offers the possibility to set breakpoints for some events:

- reinvocation of some constraint;
- labeling of a variable;
- failure;
- domain of a variable equal to some domain;
- intersection empty between the domain of a variable and a domain given previously;
- variable bound to some value.

6 Conclusion

We have presented a tool allowing to debug constraint logic programs by focusing on the store and the domain modifications propagations, along with a new propagation strategy to support it. As said previously, inspecting the store

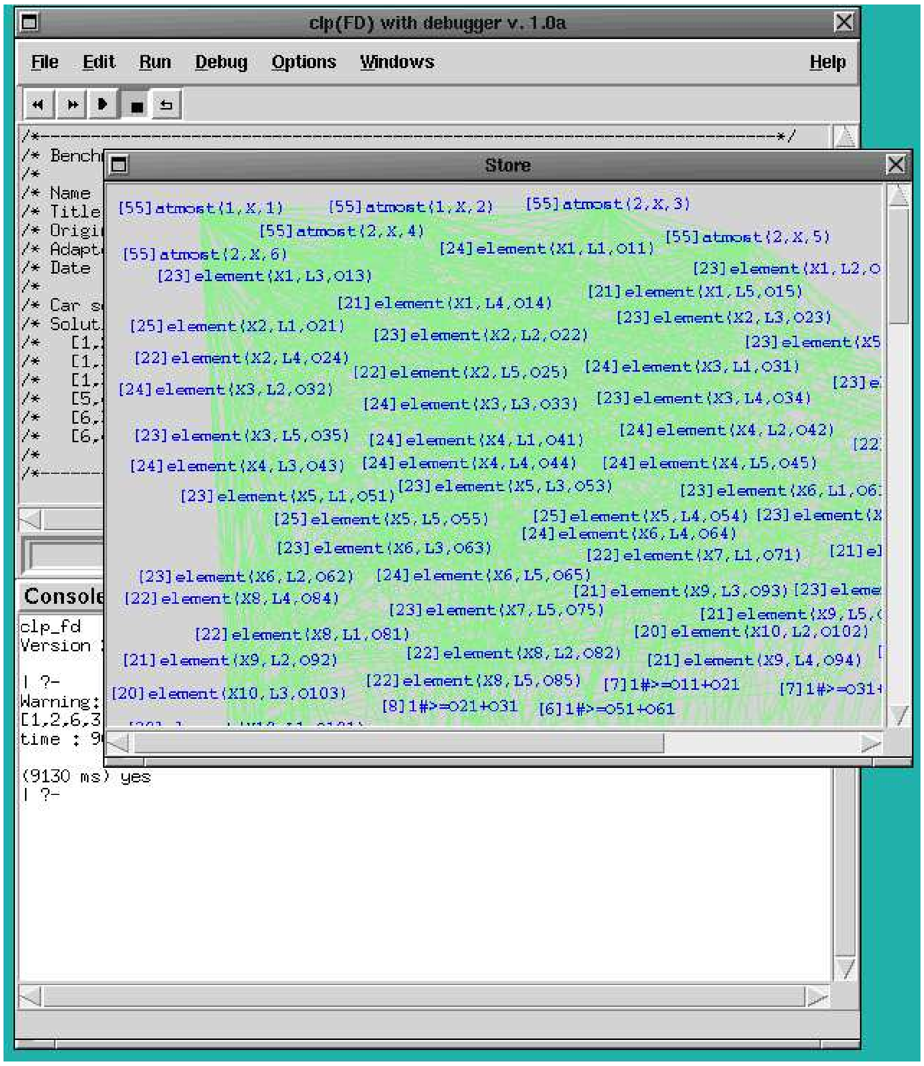

Figure 14: Resolution of `cars.pl` without S-boxes

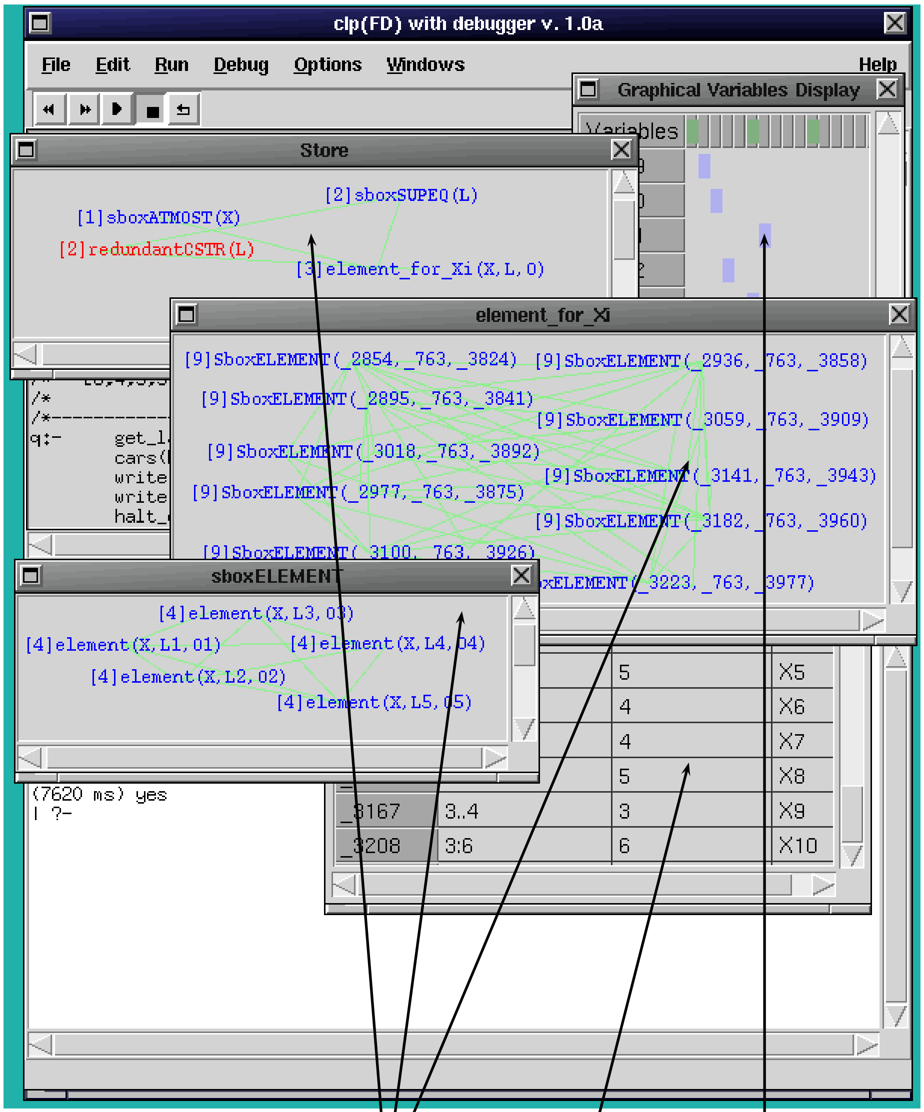

Figure 15: Resolution of `cars.pl` with S-boxes

is not sufficient, and debugging must be down at the Prolog level as well. At present, this may be done by using the text-based Prolog debugger of clp(fd). Nevertheless, we believe that the two tools are not as connected as they could be. In particular, our system lacks of a powerful graphical Prolog debugger where constraints in the source would be connected to their representation in the store. This is a planned enhancement.

The system presented here has been devised in connection with the DiSCiPl ESPRIT project on Constraint Programming Debugging. Its integration into a wider set of debugging tools is planned for the near future.

Acknowledgements

The research exposed here was supported by the reactive LTR DiSCiPl ESPRIT project N° 22532. We are indebted to all the researchers working in this project for their invaluable suggestions concerning the design of the tool described in this paper.

References

[1] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 421–435. MIT Press, 1993.

[2] Frédéric Benhamou and Frédéric Goualard. *The S-boxes based Debugger Manual*. The DiSCiPl Consortium, 1998. Not yet released.

[3] Frédéric Benhamou. Interval Constraint Logic Programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, Lectures Notes in Computer Science 10, pages 1–21. Springer Verlag, 1995.

[4] Frédéric Benhamou, Frédéric Goualard, and Laurent Granvilliers. Programming with the DecLIC Language. In *Proceedings of the Second Workshop on Interval Constraints*, 1997. Post-ILPS'97 workshop.

[5] Frédéric Benhamou and William J. Older. Applying Interval Arithmetic to Real, Integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[6] Frédéric Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea.

[7] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):252–387, October 1981.

[8] J. Burg, C. Hughes, J. Moshell, and S.D. Lang. Constraint-based programming: A survey. Technical report IST-TR-90-16, Dept. of Computer Science, University of Central Florida, 1990.

[9] C.K. Chiu and J.H.M. Lee. Efficient interval linear equality solving in constraint logic programming. Unpublished.

[10] John G. Cleary. Logical Arithmetic. *Future Generation Computing Systems*, 2(2):125–149, 1987.

[11] Philippe Codognet and Daniel Diaz. Compiling Constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[12] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[13] J. Darlington and Y.-K. Guo. A new perspective on integrating functions and logic languages. In *3rd International Conference on Fifth Generation Computer Systems*, pages 682–693, Tokyo, Japan, 1992.

[14] Daniel Diaz. *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système* `CLP(FD)` . PhD thesis, Université d'Orléans, 1995.

[15] R.E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[16] Donald E. Knuth. *Fundamental Algorithms*, volume Volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition edition, 1997.

[17] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 1(8):99–118, 1977.

[18] Micha Meier. Debugging constraint programs. In Ugo Montanari and Fracesca Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976, pages 204–221. Springer Verlag, September 1995.

[19] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[20] William J. Older and André Vellino. Constraint Arithmetic on Real Intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Papers*. The MIT Press, 1993.

[21] John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994. Seventh Printing.

[22] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94)*, Singapore, 1994.

[23] Christian Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*. The MIT Press, July 1997.

[24] I. Sutherland. *A Man-Machine Graphical Communication System*. PhD. thesis, Massachussetts Institute of Technology, January 1963.