Towards Structuring the Constraint Store

Frédéric Benhamou and Frédéric Goualard

LIFO, Université d'Orléans,
B.P. 6759, 45067 ORLEANS Cedex 2, France
`Frederic.{Benhamou,Goualard}@lifo.univ-orleans.fr`

Abstract. We present in this paper a method allowing to structure a constraint store by modifying the constraint granularity. This allows to inspect the store, which is according to us a key-point in debugging and profiling CLP programs. The method relies on properties of constraint narrowing operators and of the propagation algorithm. A theoretical framework is given, along with an example showing a use of the method.

1 Introduction

Since its introduction a decade ago, the Constraint Logic Programming paradigm has raised up a growing interest among researchers as well as industrial circles. Many implementations are by now available (e.g. `CHIP` [1], `Prolog IV` [5], `ECLiPSe`, `clp(BNR)` [4], `clp(FD)` [6], ...), and have been used to solve satisfactorily hard problems. By taking the best of two worlds (declarativity and expressiveness of Logic Programming, and efficiency of the algorithms of constraint Solving and Operational Research), CLP has proved its ability to handle real-life problems.

Nevertheless, CLP is not as widely used as it should 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 those described in [7] and [9]).

Actually, debugging (resp. profiling) CLP programs is a very difficult task. In fact, it must be done at two levels:

- at the Prolog level (debugging of the clausal structure of the program);
- at the *store* level (inspection of the constraint network).

In the following, we will focus on the store level. Inspection of the constraint network *as is* is practically intractable since the store is a huge, flat, collection of constraints with no structure —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`).

We believe that the understanding of the state of the store at each computation step is the key of constraint debugging (and profiling). It is then necessary to allow the user to control the size and complexity of the store. In the following, we will describe a method allowing the user to structure the store by modifying the constraint granularity. It relies on the properties of the *propagation algorithm* and of the *constraint narrowing operators* (presentation proposed in [2] to model the handling of constraints by solvers).

The paper is organized as follows: section 2 gives some necessary definitions extracted from [3, 2], related to the store and to the narrowing process. It extends the framework described in [3] in order to model more precisely the operational semantics of CLP solvers; section 3 introduces a notion of equivalence of *Constraint Systems* which allows us to state a recursive definition of *stores*; section 4 describes the practical spin-off of our view of what a store is; and we conclude in section 5, discussing some pending problems along with future works.

2 General framework

This section is aimed at introducing the basics of the framework modeling the operational semantics of CLP systems. We refer the reader to [3, 2] for a more general presentation.

Let Σ be a structure $< \mathcal{D}, \mathcal{F}, \mathcal{R} >$ where $\mathcal{D}$ is a set (*universe*), $\mathcal{F}$ is a set of function symbols and $\mathcal{R}$ a set of relation symbols. Let $\mathcal{V} = \{v_1, \ldots, v_p\}$ be a set of variables taking their value over $\mathcal{D}$. In this paper, constraints are conjunctions of atomic formulae built from Σ.

Informally, a constraint solver associates to every n-ary relation corresponding to a constraint a contracting monotone function (called *Constraint Narrowing operator* —CNO for short) which takes as input a Cartesian product of n domains (domains of the variables involved) and returns a Cartesian product of n domains which are smaller (or equal) to the input domains. Intuitively, a CNO eliminates from the input (some of) the values which do not belong to the underlying relation. As this cannot always be done perfectly (e.g. when dealing with relations over reals), the notion of *approximation* is introduced. Solving a constraint system corresponds then to the computation of the greatest common fixed-point of all the CNO of the system.

Definition 1 (Approximate domain) *An approximate domain $\mathcal{A}$ over a domain $\mathcal{D}$ is a subset of the power set of $\mathcal{D}$, $P(\mathcal{D})$, closed under intersection and such that $\mathcal{D} \in \mathcal{A}$.*

In practice, approximate domains considered are such that set inclusion is a well-founded relation, which guarantees termination of the algorithm considered later.

Definition 2 (Constraint Narrowing Operator) *Let Σ be a structure, $\mathcal{A}$ an approximate domain over its domain $\mathcal{D}$, and ρ an n-ary relation over $\mathcal{D}$. The function $N_\rho : \mathcal{A}^n \longrightarrow \mathcal{A}^n$ is a CNO for ρ iff the three following properties hold:*

$$\begin{array}{ll}
(1)\ \forall u \in \mathcal{A}^n : N_\rho(u) \subseteq u & \textit{(Contractance)} \\
(2)\ \forall u \in \mathcal{A}^n : u \cap \rho \subseteq N_\rho(u) & \textit{(Correctness)} \\
(3)\ \forall u, v \in \mathcal{A}^n : (u \subseteq v) \Longrightarrow (N_\rho(u) \subseteq N_\rho(v)) & \textit{(Monotonicity)}
\end{array}$$

Notations. In the following, letters X, Y, and Z refer to Cartesian products of domains; X_i refers to the i^{th} projection of X. Letters L, M, and N, refer to vectors of naturals; $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{N}$ are the ordered sets (in ascending order) of naturals induced by, respectively L, M and N; L_i (resp. $\mathbf{L}_i$) refers to the i^{th} component of L (resp. the i^{th} element of $\mathbf{L}$). Let Ω be a p-ary Cartesian product of elements of $\mathcal{A}$ such that: $\forall i \in \{1, \ldots, p\} : v_i \in \Omega_i$ (Ω is a *big box* associating each variable of $\mathcal{V}$ to its domain). We will note $\hat{\mathbf{L}}$ the cardinal of the set $\mathbf{L}$, and $\mathbf{L}^n$ the position of the natural n in the ordered set of naturals $\mathbf{L}$.

Example. Given $X = \Omega_3 \times \Omega_4 \times \Omega_5$, $L = (3, 7, 3, 5)$:

- $X_2 = \Omega_4$,
- $\mathbf{L} = \{3, 5, 7\}$,
- $L_4 = 5$,
- $\mathbf{L}_2 = 5$,
- $\mathbf{L}^5 = 2$.

□

A *constraint system* is basically a pair $E = (S, X)$ where S is a set of constraints and X is a finite Cartesian product denoting the domains of the variables appearing in S. In order to describe accurately the domain propagation step, we introduce the notion of *Extended Constraint System*[1].

Definition 3 (Extended Constraint System (ECS)) *Let Σ be a structure and $\mathcal{A}$ an approximate domain over its domain $\mathcal{D}$. An extended constraint system is a pair (S, X) where $S = \{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}$ is a set of triplets made of a k_i-ary constraint C_i, a CNO N_i for the relation associated to C_i, and a k_i-ary vector L_i collecting the indices of all the variables involved in C_i; and where X is a n-ary Cartesian product of elements of $\mathcal{A}$.*

[1] The definition given in this paper is different from the one appearing in [3] as we need a more precise view of the operational aspect.

Notations. Given an ECS $E = (S, X)$ with $S = \{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}$, we define the ordered set Λ_S collecting the indices of all the variables appearing in S as: $\Lambda_S = \mathbf{L_1} \cup \cdots \cup \mathbf{L_m}$. It is worth noticing the following property: $\forall i \in \{1, \ldots, n\} : X_i = \Omega_{\Lambda_{S_i}}$. Moreover, given two Cartesian products $X = X_1 \times \cdots \times X_n$ and $Y = Y_1 \times \cdots \times Y_m$, where $n \geq m$, and two vectors of naturals $L_X = (l_{x_1}, \ldots, l_{x_n})$ and $L_Y = (l_{y_1}, \ldots, l_{y_n})$ associated to, respectively, X and Y in such a way that: $\forall l_{x_i} \in L_X : X_i = \Omega_{l_{x_i}}$ and $\forall l_{y_i} \in L_Y : Y_i = \Omega_{l_{y_i}}$; the *intersection* of X and Y is defined as follows:

$$\forall i \in \{1, \ldots, n\} : (X \sqcap Y)_i = \begin{cases} X_i \cap Y_j & \text{if } \exists j \text{ s.t. } l_{x_i} = l_{y_j}, \\ X_i & \text{else.} \end{cases}$$

(The intersection of the components is done for all the components which match, i.e. which refer to the same domain Ω_k.)

Examples. Given
$E = \begin{pmatrix} S = \{(C_1, N_1, L_1 = (3, 2, 3)), (C_2, N_2, L_2 = (1, 5, 3, 7)), (C_3, N_3, L_3 = (2)), \}, \\ X = \Omega_1 \times \Omega_2 \times \Omega_3 \times \Omega_5 \times \Omega_7 \end{pmatrix}$:

- $\Lambda_S = \mathbf{L_1} \cup \mathbf{L_2} \cup \mathbf{L_3} = \{1, 2, 3, 5, 7\}$
- given $X = (\{3, 4\}, \{2, 5, 1\}, \{1, 3\})$, $L_X = (2, 5, 4)$, $Y = (\{9\}, \{3\})$ and $L_Y = (5, 2)$:

$$X \sqcap Y = (\{3\}, \emptyset, \{1, 3\})$$

□

Given the triplet (C, N, L) where C is a k-ary constraint, the greatest fixed-point of N, $fp(N)$, is a k-ary vector of domains. Let $\mathbf{fp(N)}$ be the set of domains obtained from $fp(N)$ in such a way that: $\forall j \in \{1, \ldots, k\} : fp(N)_j = \mathbf{fp(N)}_{\mathbf{L}^{L_j}}$ (that is, $\mathbf{fp(N)}$ is the set of domains ordered according to the indices of the variables in $\mathbf{L}$). We will note $\underline{fp(N)}$ the n-ary vector of domains defined by:

$$\forall i \in \{1, \ldots, n\} : \underline{fp(N)}_i = \begin{cases} \mathbf{fp(N)}_{\mathbf{L}^{\Lambda_{S_i}}} & \text{if } \Lambda_{S_i} \in \mathbf{L}, \\ \mathcal{D} & \text{else.} \end{cases}$$

The declarative semantics of an ECS E is the set $E^\star$ of n-ary tuples from X satisfying the conjunction of the constraints $C_1, \ldots, C_m$. Since $E^\star$ is generally not computable, we will use an approximate semantics $\overline{E}$ of E instead, which is the greatest common fixed-point of the N_i's included in X, that is:

$$\overline{E} = max(\{u \in \bigcap_{i=1}^{m} \underline{fp(N_i)} \mid u \subseteq X\})$$

Definition 4 (Weak arc-consistency) *Let $E = (\{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}, X)$ be an extended constraint system. E is weakly arc-consistent iff $X = \overline{E}$.*

Given an ECS $E = (S, X)$, we use the algorithm NAR described in table 1 in order to compute an ECS $E' = (S, X')$ which is *maximally weak arc-consistent* w.r.t. X (i.e. X' is the greatest Cartesian product of domains included in X such that E' is weak arc-consistent).

We give without proof (see [3, 8]) the proposition below stating some of the properties of NAR to be used later:

Proposition 1 (Narrowing algorithm) NAR *is correct, confluent, contractant and monotone.*

```
NAR( in {(C_1, N_1, L_1), ..., (C_m, N_m, L_m)} ; inout X = X_1 × ... × X_n )
begin
    P := {C_1, ..., C_m}
    while (P ≠ ∅) and (∀k ∈ {1, ..., n} : X_k ≠ ∅) do
        choose one C_i in P
        (C, N, L) := (C_i, N_i, L_i)
        Y := X_{L_1} × ··· × X_{L_L̂}
        X' := X ⊓ N(Y)
        if X' ≠ X then
            P := P ∪ {C_j | ∃k ∈ L_j ∧ X'_k ≠ X_k}
            X := X'
        endif
        P := P \ {C}
    endwhile
end.
```

Table 1. The Narrowing algorithm.

3 Structuring the store

Notations. We introduce two new notations to be used in the following:

1. *Restriction.* We define the *restriction* of a Cartesian product X to a triplet $CN = (C, N, L)$ to be:
$$X|_{CN} = \Omega_{\mathbf{L}_1} \times \cdots \times \Omega_{\mathbf{L}_{\hat{L}}}$$
2. *Union.* Given $E = (S, X)$, $E' = (S', X')$, and $\Lambda = \Lambda_S \cup \Lambda_{S'}$, the *union* of X and Y is defined by:
$$X \bowtie Y = \Omega_{\Lambda_1} \times \cdots \times \Omega_{\Lambda_{\hat{\Lambda}}}$$
3. *Expansion.* Given $L = (L_1, \ldots, L_l)$ and $M = (M_1, \ldots, M_m)$, we will note $L \oplus M$ the vector obtained by "concatenating" L and M:
$$L \oplus M = (L_1, \ldots, L_l, M_1, \ldots, M_m)$$

□

In the framework described above, a *store* is an ECS

$$E = (S, X), \text{ where } S = \{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\} \tag{1}$$

Moreover, the classical view is to consider that the store is the constraint network gathering all the constraints which have been stated so far. We will see in the following that it is best considering the store as representing only a subset of all constraints already stated.

An immediate corollary of proposition 1 is that, given an ECS E of the form (1), $\textsc{Nar}(E)$ is a constraint narrowing operator for $C_1 \wedge \ldots \wedge C_m$. It is then possible to define ECSs of the form (1) where some $CN = (C', N', L') \in S$ are triplets of the form:

$$\begin{cases} C' = C'_1 \wedge \ldots \wedge C'_{m'} & (2) \\ N' = \textsc{Nar}((\{(C'_1, N'_1, L'_1), \ldots, (C'_{m'}, N'_{m'}, L'_{m'})\}, X|_{CN})) & (3) \\ L' = L'_1 \oplus \cdots \oplus L'_{m'} & (4) \end{cases}$$

We obtain thus a definition of ECSs which is "recursive":

Definition 5 (Hierarchical Extended Constraint System (HECS)) *Let Σ be a structure and $\mathcal{A}$ an approximate domain over its domain $\mathcal{D}$. A hierarchical extended constraint system is a pair (S, X) where $S = \{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}$ is a set of triplets (C_i, N_i, L_i) where C_i is an atomic formula or a conjunction of the form (2), N_i is a CNO —being possibly of the form (3) when C_i is of the form (2)—, and L_i is a vector of naturals; and where X is a n-ary Cartesian product of elements of $\mathcal{A}$.*

Note that every ECS is a HECS. Thanks to the confluency of NAR, it is possible to gather arbitrarily some constraints $C_a, \ldots, C_k$ of a (H)ECS E into a constraint $C_\alpha = C_a \wedge \ldots \wedge C_k$ (with its associated narrowing operator $N_\alpha = \text{NAR}((\{(C_a, N_a, L_a), \ldots, (C_k, N_k, L_k)\}, X|_{CN_a} \bowtie \cdots \bowtie X|_{CN_k}))$ and its vector $L_\alpha = L_a \oplus \cdots \oplus L_k$) such that the HECS E' obtained from E by replacing the triplets $(C_a, N_a, L_a), \ldots, (C_k, N_k, L_k)$ by $(C_\alpha, N_\alpha, L_\alpha)$ has the same declarative semantics than E.

More formally, given a HECS E, let UNFOLD(E)=$\tilde{E}$ be the output of the algorithm described in table 2 (intuitively, UNFOLD(E) flattens E such that there is no longer any CNO of the form (3) in it); we state without proof the proposition below:

Proposition 2 (Equivalence of HECSs) *Given two HECSs E and E', the following implication holds:*

$$\tilde{E} = \tilde{E'} \Longrightarrow \overline{E} = \overline{E'}$$

Let N_{NAR} be the set of CNO of the form (3).

UNFOLD$\big((S, X)\big)$=
begin
  NAC $\leftarrow \{(C_i, N_i, L_i) \in S \mid N_i \in N_{\text{NAR}}\}$
  S' $\leftarrow \emptyset$
  forall $\big(\beta = (C_{\beta_1} \wedge \ldots \wedge C_{\beta_k}, \text{NAR}(\{(C_{\beta_1}, N_{\beta_1}), \ldots, (C_{\beta_k}, N_{\beta_k})\}, X|_\beta), L_{\beta_1} \oplus \cdots \oplus L_{\beta_k})\big) \in$ NAC **do**
    $(S_\beta, X|_\beta) \leftarrow$ UNFOLD$((\{(C_{\beta_1}, N_{\beta_1}, L_{\beta_1}), (C_{\beta_k}, N_{\beta_k}, L_{\beta_k})\}, X|_\beta))$
    S' $\leftarrow$ S' $\cup\, S_\beta$
  endforall
  forall $\big(\gamma = (C_\gamma, N_\gamma, L_\gamma)\big) \in S\backslash$NAC **do**
    S' $\leftarrow$ S' $\cup \{\gamma\}$
  endforall
  return(S',X)
end.

Table 2. The UNFOLD algorithm.

This proposition implies that it is possible to structure the global store (containing all constraints) in a hierarchical way, into a conjunction of sub-stores (gathering only some of the constraints), while preserving the declarative semantics of the whole. Each sub-store acts exactly as a global store. The idea is then to let the user focus only on the sub-store he is interested in, throwing temporarily away all the constraints not included in it. We will develop this idea in the next section.

4 The store-as-a-box view

We introduce now the notion of *S-box* [2] which will be used throughout the rest of the paper:

[2] *"S"* stands for *"Store"*.

Definition 6 (S-box) *A S-box $B = \langle T, X\rangle$ is a pair composed of a triplet $T = (C, N, L)$, where C is an atomic formula or a conjunction of formulae, N is a CNO, and L is a vector of naturals; and of a Cartesian product of domains $X = X_1 \times \cdots \times X_k$ (k being equal to the number of variables occurring in C).*

Note that there exists a trivial correspondence between HECS and S-boxes, as a HECS $E = (\{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}, X)$ may be viewed as the S-box

$$B = \big\langle \big(C_1 \wedge \ldots \wedge C_m, \textsc{Nar}((\{(C_1, N_1, L_1), \ldots, (C_m, N_m, L_m)\}, X)), L_1 \oplus \cdots \oplus L_m\big), X\big\rangle$$

and vice-versa.

A S-box is a black-box communicating with the "outside" only through its input/output (namely X).

Given a S-box $B = \big\langle (C, N, L), X\big\rangle$, let $\textsc{Expand}(B)$ be the set of S-boxes defined as:

$$\textsc{Expand}(B) = \begin{cases} \emptyset & \text{if } N \notin N_{\text{NAR}}; \\ \{\langle (C_1, N_1, L_1), X|_{CN_1}\rangle, \ldots, \langle (C_m, N_m, L_m), X|_{CN_m}\rangle\} & \text{if } C = C_1 \wedge \ldots \wedge C_m, \\ & N{=}\,\textsc{Nar}((\bigcup_{i=1}^{m} \{(C_i, N_i, L_i)\}, X)), \\ & \text{and } L = L_1 \oplus \cdots \oplus L_m \end{cases}$$

Given a S-box $B = \big\langle (C, N, L), X\big\rangle$, let T_B be the tree "induced" by B: informally, each node NT_B of T_B is a S-box B_{NT_B} and possesses as many children as there exists sub-S-boxes in B_{NT_B}. For the root of T_B, we have:

$$\begin{cases} \text{root}(T_B) = B \\ \text{children}(\text{root}(T_B)) = \Big\{ \big(B_i = \langle (C_i, N_i, L_i), X|_{CN_i}\rangle\big) \in \textsc{Expand}(B) \mid N_i \in N_{\text{NAR}} \Big\} \end{cases}$$

The same definition holds for each child of root(T_B) (which is itself a S-box). The figure 1 gives an example of the correspondence S-box/tree along with the notation used to label each node of the tree.

Given a tree T, we will note $T|_p$ the S-box being at position p in T (e.g. in figure 1, $T_B|_1 = \langle (C_4 \wedge C_5, N_2, L_2), X|_{CN_2}\rangle$).

We then define the notion of *global S-box* associated to a tree T as being the S-box $T|_\varepsilon$. For a computation of a CLP program, the global S-box relates to what is usually called *store* (that is, it embedds all the constraints stated so far). With our model, we have also the central notion of *current location* which corresponds intuitively to the position in the tree T of the S-box the user is currently focusing on. Knowing that S-boxes are black boxes, it appears that for a given current location CL and a tree T, all constraints but those embedded in S-boxes belonging to $T|_{CL}$ are "forgotten" when propagating domain modification of the variables. From now on, we will call *store* the S-box located at $T|_{CL}$.

4.1 Description of the tool

In order to be useful, our device must provide at least the ability to:

- create S-boxes;
- modify the current location (i.e. focus in and out).

We plan to allow creation of S-boxes following two ways:

1. by tagging some goals in a CLP program (all constraints added to the store while executing the goals are gathered in one S-box);

Given $\{C_1, C_4, C_6, C_7, C_8, C_9, C_{10}\}$ s.t. $\{N_1, N_4, N_6, N_7, N_8, N_9, N_{10}\} \not\subset N_{\mathrm{NAR}}$ and the relations $C_5 = C_6 \wedge C_7$, $C_2 = C_4 \wedge C_5$ and $C_3 = C_8 \wedge C_9 \wedge C_{10}$; let

$$B = \langle (C_1 \wedge C_2 \wedge C_3, N_0, L_0), X \rangle$$

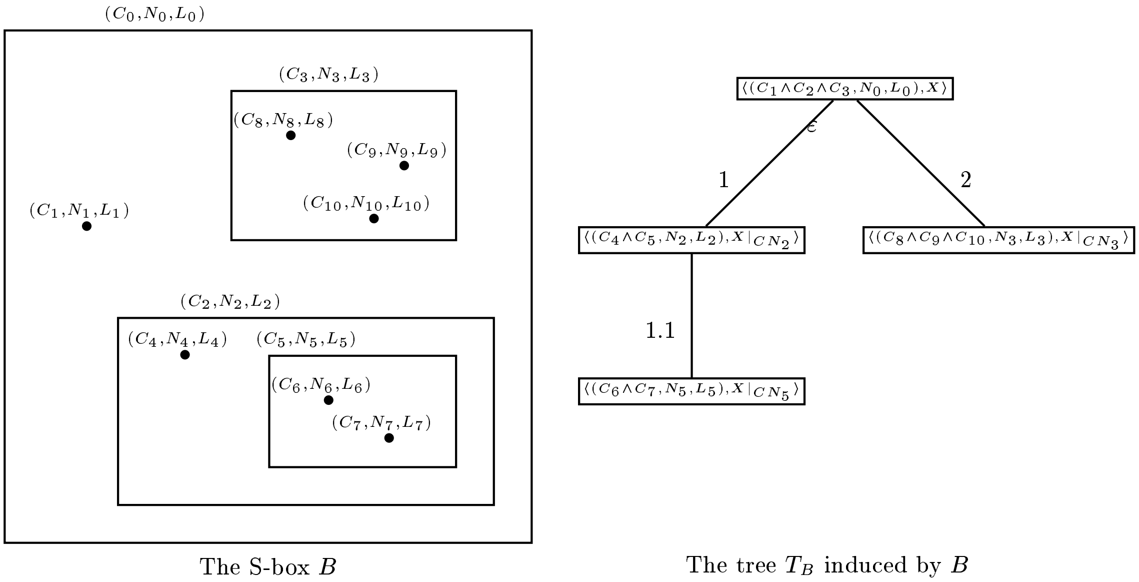

Fig. 1. Correspondence S-box/tree.

2. by a direct manipulation on a graphical display of the store along with the existing S-boxes (manipulation allowed will be such that the hierarchy of S-boxes will remain a tree).

We will give in the remaining of the section an example of use of the tool.

Let consider the following CLP program P aimed at finding the time T at which a ball moving along a curve will hit a wall.

```
object_A(X,Y,Z):- % Shape of the wall.
        X ≤ 0,
        Y ≤ 0,
        Z ≤ 0.
object_B(XCx,YCy,ZCz):- % Shape of the ball.
        XCx² + YCy² + ZCz² = 1.
center_B(T,Cx,Cy,Cz):- % Pos. of the center of the ball at T.
        T² − Cx = 10,
        2T − Cy = 10,
        Cz − T² + 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 would be to tag all the goals, 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. It simplifies greatly the store by masking many constraints while preserving the declarative semantics, as said previously. For example, we could view the store of the figure 2 as the (simplified) store of figure 3. Note that the number of links between

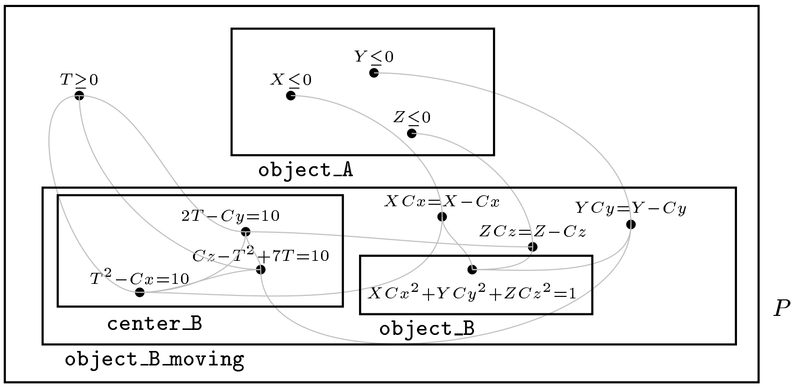

Fig. 2. Store structure for program P when tagging all goals.

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 while propagating domains from a S-box to an other S-box. For example, assume that the propagation of some domains given by the user as an input to the box of $R_4(X, Y, Z, T)$ do not lead to the right domains as output; in order to find what has gone wrong, the user can focus on the S-box of $R_4(X, Y, Z, T)$, throwing away temporarily all the other constraints. This corresponds to a modification of the *current location* (see above).

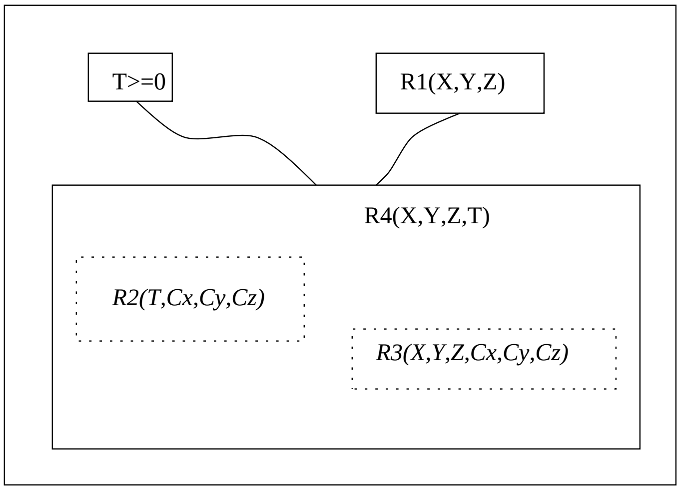

Fig. 3. Abstraction for the store of the program P.

Focusing on $R_4(X, Y, Z, T)$, we obtain the store described in figure 4. From this point, it is even possible to go farther and to focus on the S-boxes of $R_2(T, C_x, C_y, C_z)$ and $R_3(X, Y, Z, C_x, C_y, C_z)$ until the source of the problem is found. Note that it is also possible to split an S-box in two or more S-boxes in order to refine the search.

Our tool is intended to be graphical and would display the store in a similar way as in figure 2. Note that, in order to keep all simple, there is no problem of variable naming in our example. The user would have the possibility to change the *current location* by clicking on a S-box, which would promote it to be the store (refer to what have been said above), or by

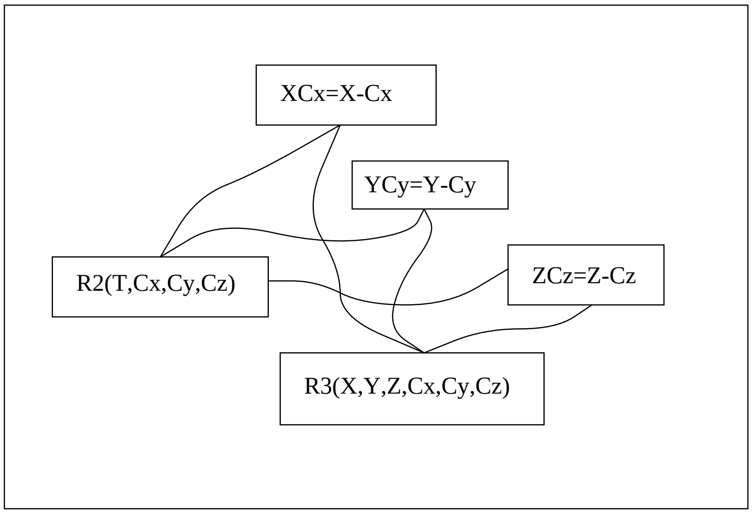

Fig. 4. The new store after focusing on $R_4(X, Y, Z, T)$.

selecting in a window displaying the tree induced by the hierarchy of S-boxes the node he is interested in. By creating a suitable hierarchy of S-boxes, the user would be allowed to "forget" some constraints known to be correct or some parts of his CLP program he does not want to inspect for the moment (a property of some S-boxes belonging to the *store* could be a *maskable ability*, that is, masked S-boxes in the store would be involved in the propagation step but would act exactly as primitive constraints (no tracing in them)).

Note that S-boxes may be used in order to handle complex constraints as well as constraints decomposed into primitive ones (as is done in many CLP systems) in a unified way. Moreover, it would be possible to display a complex constraint as a whole (i.e. as stated in the CLP program) or as the conjunction of primitive constraints —created by the solver— embedded in a S-box, at the user will. However, note that the two are not exactly equivalent, as the introduction of a S-box modifies the propagation step.

5 Conclusion and future works

We have presented a method allowing to structure a constraint store. We believe that the integration of this method in a graphical tool will permit to debug and profile CLP programs by focusing only on some subsets of constraints (e.g. those responsible for an inconsistence). As it can be seen, it is somewhat possible to relate a structuration of a store to a new CLP program (each S-box being modeled by a clause). Thus, a promising direction of research would be to adapt the work done on declarative debugging to our scheme at hand.

Nevertheless, some problems are still to be tackled, one of them being the intuitive naming of variables when representing graphically the store. Another problem is to represent in an intuitive way some complex constraints which are created recursively (e.g. the sum of some expressions being equal to 0).

Finally, the biggest problem yet to be solved is to find a way to define how to construct the best hierarchy of S-boxes for a program at hand, for we think that our method needs to rely on a —partially— automatized process in order to allow the user to debug real-life programs.

Acknowledgements

The research exposed here is supported by the reactive LTR n° 22532 DiSCiPl ESPRIT Project.

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. F. 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.
3. F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.
4. F. Benhamou and W. J. Older. Applying Interval Arithmetic to Real, Integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
5. 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.
6. Ph. Codognet and D. Diaz. Compiling Constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.
7. 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.
8. W. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Papers.* The MIT Press, 1993.
9. 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.