EXPERIMENTS IN REACTIVE CONSTRAINT LOGIC PROGRAMMING

FRANCOIS FAGES, JULIAN FOWLER and THIERRY SOLA

▷ In this paper we study a reactive extension of constraint logic programming. Our primary concerns are search problems in a dynamic environment, where interactions with the user (e.g. in interactive multi-criteria optimization problems) or interactions with the physical world (e.g. in time evolving problems) can be modeled and solved efficiently. Our approach is based on a complete set of query manipulation commands for both the addition and the deletion of constraints and atoms in the query. We define a fully incremental model of execution which, contrary to other proposals, retains as much information as possible from the last derivation preceding a query manipulation command. The completeness of the execution model is proved in a simple framework of transformations for CSLD derivations, and of constraint propagation seen as chaotic iteration of closure operators. A prototype implementation of this execution model is described and evaluated on two applications. ◁

1. INTRODUCTION

The integration of constraint programming and logic programming resulted in a powerful model of computation that is conceptually simple and semantically elegant [16]. Constraint logic programming (CLP) systems have been proved successful in a wide variety of complex system modeling and combinatorial optimization problems. Numerous applications have been developed over the last decade across various

Address correspondence to F. Fages, CNRS, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France. `fages@dmi.ens.fr` and T. Sola, Thomson-CSF Communications, 160 Bd Valmy, 92704 Colombes, France, France.

This paper is the complete version of a previous paper published in [14].

application domains, ranging from options trading and financial planning to job-shop scheduling, crew management, etc. [17].

A promising way to enlarge these domains of application is to generalize the CLP paradigm for dealing efficiently with open systems whose objective is not to produce a single input-output relation but to maintain an interaction with the environment. This class of systems has been called reactive systems by Harel and Pnueli [15]. The reactive CLP systems we consider may not have strong response time requirements but we do want to model their behavior over time, and provide them with an efficient incremental execution model.

The capacity to interact with the environment is indispensable in any system, whether this interaction is with users, sensors or effectors. This capacity may become preponderant in some domains. Our experience concerns search problems in a dynamic environment, more especially of one or both of the following kinds:

- Decision support systems where the interaction with the user is a fundamental property. Interactive decision support systems allow a much more powerful form of problem solving than their non-interactive counterparts. The solution presented by the system is just a reference point in the interactive elaboration of a final decision. The user can thus continue to interact in order to define his requirements by successive approximation. This is especially pertinent for multi-criteria optimization where the knowledge on the combination of criteria which constitutes a good solution is necessarily partial and context-dependent.

- On-line planning, scheduling and resource allocation problems where it is necessary to modify the solution to account for new information. For example while executing a schedule, a problem such as machine failure may arise, thus requiring the revision of the current schedule.

The realization of reactive and interactive systems within the CLP framework requires that the model of execution be extended with a mechanism to capture external events. Concurrent constraint (CC) programming [26] extends CLP with one form of communication, synchronization and data-driven computations, based on constraint entailment (*ask* operation). However, the monotonicity hypothesis (i.e. information is accumulated in the store of constraints but never removed) does not fit well with open systems. Recent proposals have been made to palliate this drawback, by considering timed CC programs [25], non-monotonic and linear CC programs [10] [6]. Our work belongs to a similar line of research, with a strong emphasis on the practicality of the scheme and its evaluation on some real-world applications, but with a different focus on *search problems* in a dynamic environment, whereas in the CC approaches to reactive systems, non-determinism is usually replaced by committed-choice indeterminism. Also we shall not consider *ask* operations and will stick to the CLP scheme.

In CLP, one possible choice is to consider external events as query modification commands. Maher and Stuckey [22] defined an incremental execution model only for the addition of atoms and constraints to the query. Van Hentenryck [29] described methods of re-execution by oracle for both the addition and the deletion of constraints to a query. Neither of these methods, however, offers all the possibilities of incremental addition and deletion of constraints and atoms to the query.

In this paper we study a reactive execution model for CLP which allows all query manipulation commands. Contrary to the re-execution models where several derivations are memorized, our model only retains the last derivation preceding a

query manipulation command. After an interaction, the resolution of atoms causing a failure together with a newly added constraint are delayed in such a way that as much information as possible from the previous derivation is retained. The method relies both on a system of transformations of CSLD derivations and on the capability of the constraint solver to efficiently deal with addition and deletion of constraints. We present a generic incremental constraint propagation algorithm in an abstract framework where constraints are identified with closure operators in a lattice of variable environments. This presentation allows a simple proof of correctness of the incremental constraint solver.

The plan of the paper is the following. The next section fixes preliminaries and notations on CLP languages. Section 3 presents the hypotheses under which our scheme is applicable to reactive systems, followed by the formal definition of the system of transformations of CSLD derivations. A general incremental constraint propagation algorithm that supports arbitrary addition and deletion of constraints is presented in section 4. The base of our implementation is then described in section 5 with a meta-interpreter which combines the incremental constraint propagation algorithm with a reactive search procedure. Sections 6 and 7 present evaluation results on two applications: an interactive decision support system for frequency band allocation, and an on-line aircraft sequencing problem in a simulated environment. Section 8 gives some comparisons with other works.

2. PRELIMINARIES AND NOTATIONS ON CLP LANGUAGES

This presentation conforms where possible to [17] [16]. A constraint language is defined on a signature Σ of constants, functions and predicate symbols (containing *true, false* and $=$), and on a countably infinite set $\mathcal{V}$ of variables. An *atomic constraint*, noted $a_1, a_2...$, has the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol in Σ and the t_i's are Σ,$\mathcal{V}$-terms. A *constraint*, noted c, $d...$, is a conjunction of atomic constraints. The set of variables of a constraint c is noted $\mathcal{V}(c)$. Syntactically constraints will also be seen as finite multisets of atomic constraints, where the *multiset union* of constraints c and d noted c, d denotes the conjunction of constraints, and *multiset difference*[1] is noted $c \setminus d$ (*true* denotes the empty multiset of constraints).

A mathematical structure $\mathcal{D}$ is assumed to fix the interpretation of constraints. The $\mathcal{D}$-satisfiability of constraints is assumed to be decidable, i.e. one can decide for any constraint c whether $\mathcal{D} \models \exists X\ c$ or $\mathcal{D} \models \neg\exists X\ c$ where $X = \mathcal{V}(c)$.

$CLP(\mathcal{D})$ programs are defined with an extra signature Π of predicate symbols disjoint from Σ. An *atom* has the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol in Π and the t_i's are Σ,$\mathcal{V}$-terms. A definite CLP program is a finite set of clauses of the form $(A \leftarrow c \mid \alpha)$, where A is an atom, c is a constraint, and α is a finite multiset of atoms ($\Box$ denotes the empty multiset of atoms). A goal is noted $c|\alpha$ where c is a constraint and α a multiset of atoms. In the rest of the paper we assume without loss of generality that the programs and goals are in *canonical form*, where the atoms are formed with variables only, constant and function symbols appear exclusively in constraints. For example the program clause $p(X+1) \leftarrow p(X-1)$ should be written in canonical form $p(Y) \leftarrow Y = X+1, Z = X-1 | p(Z)$.

[1] Formally, a multiset c (a constraint here) of elements from some universe $\mathcal{A}$ (the set of atomic constraints here), is an application from $\mathcal{A}$ to $\mathbf{N}$, that indicates the number of occurrences in c of each element $a \in \mathcal{A}$. Multiset union is defined by $(c, d)a = c(a) + d(a)$. The multiset difference considered here is defined by $(c \setminus d)a = max\{0, c(a) - d(a)\}$.

Operationaly, CLP($\mathcal{D}$) programs are interpreted by a simple transition system defined by the CSLD resolution rule. For our purpose it is convenient to represent explicitly *failed goals* with an inconsistent constraint. Therefore the test of satisfiability of the constraints is on the goal to rewrite, not on the resulting goal which may be inconsistent.

$$CSLD\frac{(A\leftarrow d|\beta) \in P \quad \mathcal{D} \models \exists c}{c|A,\alpha \overset{A\leftarrow d|\beta}{\longrightarrow} c,d|\beta,\alpha}$$

The atom A in the transition is called the *selected atom*. A *CSLD derivation* is a sequence of CSLD transitions written $G_0 \overset{A_1\leftarrow c_1|\alpha_1}{\longrightarrow} G_1 \overset{A_2\leftarrow c_2|\alpha_2}{\longrightarrow} ...$, or simply $G_0 \longrightarrow G_1 \longrightarrow ...$ when the rules applied are clear from the context. A derivation is *successful with answer constraint* c if it is finite and ends with a goal of the form $c|\Box$ where c is $\mathcal{D}$-satisfiable. A derivation is *finitely failed* if it is finite and ends with a goal with a $\mathcal{D}$-unsatisfiable constraint. A *CSLD tree* for a goal G is the tree of all derivations from G obtained by fixing a selected atom in each node.

Theorem 2.1 (Soundness and Completeness of CSLD resolution). [16] [23]
Let P be a CLP($\mathcal{D}$) program and G a goal. Let T be a CSLD tree for G. If T contains a successful derivation with answer constraint c then $P,\mathcal{D} \models c \rightarrow G$. If $P,\mathcal{D} \models c \rightarrow G$ then there exist successful derivations in T with answer constraints $c_1,...,c_n$ such that $\mathcal{D} \models c \rightarrow \exists Y_1 c_1 \vee ... \vee \exists Y_n c_n$ where $Y_i = \mathcal{V}(c_i) \setminus \mathcal{V}(c)$.

3. AN INCREMENTAL EXECUTION MODEL FOR REACTIVE CLP

3.1. Hypotheses for Reactive Constraint Logic Programming

In this section the hypotheses underlying and justifying our approach are presented. The explication of these hypotheses serves as an informal description of the execution model presented in the next section.

Hypothesis 1. Interactions from the environment only modify the top-level goal.

This principle hypothesis states that all interactions with the environment go through the top-level goal and computed answer constraints. The four basic goal transformation commands are the addition and removal of a constraint or an atom. The syntax of these commands is given in table 1). The novelty with the primitives of Maher and Stuckey for query manipulation [22] is the *deletion* of constraints and atoms.

del_constraint(c)
add_constraint(c)
del_atom(A)
add_atom(A)

Table 1. Syntax of basic goal manipulation commands

A consequence of hypothesis 1 is that the data that is subject to change must be contained in the goal, not the program clauses. Here we do not distinguish between

a modification due to the interaction of a user (interactive system) or of an arbitrary external agent (reactive system). We shall see that the interactions are allowed at any point in the CSLD resolution process.

The basis of the procedure is to use the information contained in the CSLD tree for the preceding goal to reduce the computation required for the modified goal. When an interaction modifies the goal, the associated CSLD tree is transformed into a new partial CSLD tree for the new goal. Of course certain parts of the preceding partial CSLD tree are removed in this transformation but others remain valid. This operational intuition forms the basis of previous schemes for goal modification presented in [3, 22, 29]. We insist, however, that the space complexity of the information we retain is independent of the number of interactions.

Hypothesis 2. Space complexity is independent of the number of interactions.

Unlike [29], our execution model thus conserves only the information contained in the partially constructed CSLD tree of the preceding goal. Furthermore, unlike [22], the transformation of the CSLD tree is based on a single derivation for the preceding goal. This choice leads to a new execution model for constraint logic programming instead of a purely meta-level extension.

The CSLD tree transformations try to preserve as many steps of the previous derivation as possible. This has a double effect. It minimizes the re-execution necessary to search for new solutions, and the scheme is more likely to enumerate solutions which are close in some sense to the solution found previously. However for sparse problems in which all subproblems are strongly connected the search for a new solution may necessitate the revision of the totality of a derivation. In that case re-execution from scratch may be more efficient than an incremental scheme. Therefore our last (loose) hypothesis is about the structure of the problems that the scheme is best suited to solve.

Hypothesis 3. Dense problems are considered, so that small changes in a goal result in revising only a few number of steps in the derivation for a new solution.

We shall see that this hypothesis can be made technically more precise in terms of the number of connected components in the dynamic dependency graph of constraints. The capability of the scheme to re-order the selected atoms in a CSLD derivation makes it possible to re-use a large part of the previously successful derivation and to limit the search space for new solutions to few subgoals. This capability will also be used to define new search procedures for static CLP problems.

The execution model of the reactive constraint logic programming scheme is presented in two parts. The first part defines the transformations of CSLD trees after an interaction. The second part presents the requirements for the constraint solver, which is described in details in the next section. Then a discussion of the associated search procedures is given in the following section on implementation.

3.2. Transformations of CSLD Trees

When an interaction occurs, the CSLD tree for the current goal has been searched up to a certain point defined by a single derivation. The transformation of the CSLD tree for the modified goal is based on the transformation of that derivation.

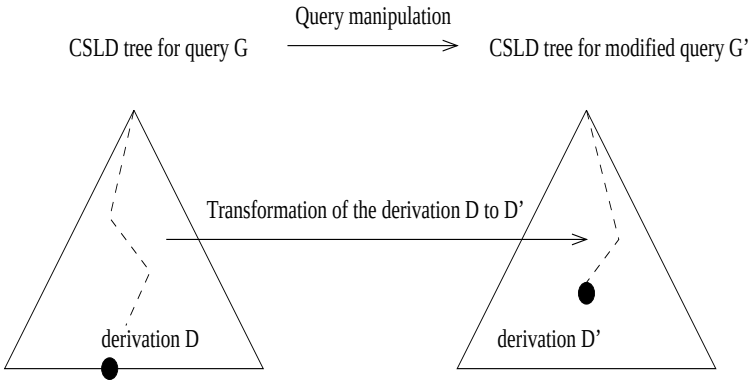

Figure 1. Transformation of a CSLD tree based on one derivation.

Two basic operations are defined on CSLD derivations: the pruning of a derivation by a constraint (i.e. the addition of the constraint to the goals of the derivation) and its complement the lifting of a derivation (the deletion of a constraint from the goals of the derivation). Similar operations are defined for atoms: the addition of a multiset of atoms to the initial goal of a CSLD derivation, and its somewhat more complex counterpart, the removal of a multiset of atoms from the initial goal of a derivation.

Definition 3.1. The pruning of a CSLD derivation Δ by a constraint c is the derivation, noted $\Delta \otimes c$, obtained by adding c to the goals in Δ up to inconsistency:

$$\begin{array}{lcl} (d|\beta) \otimes c & = & (c, d|\beta), \\ (d|\beta \longrightarrow \Delta') \otimes c & = & (c, d|\beta) \longrightarrow (\Delta' \otimes c) \text{ if } c \wedge d \text{ is satisfiable,} \\ & & (c, d|\beta) \text{ otherwise.} \end{array}$$

The lifting of a CSLD derivation Δ by a constraint c supposed to occur in the initial goal is the derivation, noted $\Delta \oslash c$, obtained by deleting c in the derivation:

$$\begin{array}{lcl} (c, d|\alpha) \oslash c & = & (d|\alpha), \\ (c, d|\alpha \longrightarrow \Delta') \oslash c & = & (d|\alpha \longrightarrow (\Delta' \oslash c)). \end{array}$$

The addition of atoms is defined similarly. The deletion of atoms in a derivation must take care of the dependencies between atoms in the CSLD derivation.

Definition 3.2. The addition of atoms α to a CSLD derivation Δ, is the derivation, noted $\Delta \oplus \alpha$, obtained by adding atoms α to the goals in Δ:

$$\begin{array}{lcl} (c|\beta) \oplus \alpha & = & (c|\alpha, \beta), \\ (c|\beta \longrightarrow \Delta') \oplus \alpha & = & (c|\alpha, \beta \longrightarrow (\Delta' \oplus \alpha)). \end{array}$$

The deletion of atoms α in a CSLD derivation Δ whose initial goal contains atoms α, is the CSLD derivation, noted $\Delta \ominus \alpha$, defined by:

$$\begin{array}{lcl} (c|\alpha, \beta) \ominus \alpha & = & (c|\beta), \\ (c|\alpha, \beta \stackrel{A \leftarrow d|\gamma}{\longrightarrow} \Delta') \ominus \alpha & = & (c|\beta \stackrel{A \leftarrow d|\gamma}{\longrightarrow} (\Delta' \ominus \alpha)) \text{ if } A \notin \alpha, \\ & = & \Delta' \oslash d \ominus \alpha' \text{ if } A \in \alpha \text{ and } \alpha' = (\alpha \setminus \{A\}) \cup \gamma. \end{array}$$

Note that the operation of pruning by a constraint does not change the order of selected atoms along the derivation. In order to preserve a maximum number of deductions from the previous CSLD derivation it is possible to delay the selection

of an atom which introduces an inconsistency, instead of cutting the derivation at the first encountered inconsistency. The following operation formalizes this idea, it marks the fundamental difference between our method and the methods of [22] and [29] for the constraint addition command defined below.

Definition 3.3. The delaying of the resolution steps which introduce a constraint c in a derivation Δ is the derivation, noted $\Delta \odot c$, defined recursively by:

$$\begin{array}{lcl} (d|\alpha) \odot c & = & (d|\alpha), \\ (d|\alpha \overset{A \leftarrow e|\beta}{\longrightarrow} \Delta') \odot c & = & (d|\alpha \overset{A \leftarrow e|\beta}{\longrightarrow} (\Delta' \odot c)) \text{ if } e \neq c, \\ & = & (\Delta' \oslash c \ominus \beta \oplus \{A\}) \odot c \text{ if } e = c. \end{array}$$

Now the goal manipulation commands can be defined formally as operators acting on couples noted between angle brackets $< G \ ; \ \Delta >$, formed of a goal and a derivation.

Definition 3.4. Let G be a goal and Δ be a CSLD derivation from G. The goal manipulation commands are defined by the following transformations:

$< c, d\|\alpha \ ; \ \Delta >$	*del−constraint(c)*	$< d\|\alpha \ ; \ \Delta \oslash c >$
$< d\|\alpha \ ; \ \Delta >$	*add−constraint(c)*	$< c, d\|\alpha \ ; \ \Delta \otimes c >$ *if $e \wedge c$ is satisfiable where $e\|\gamma$ is the final goal in Δ*
		$< c, d\|\alpha \ ; \ \Delta \odot c_1 ... \odot c_n \otimes c >$ *if $\{c_1, ..., c_n\}$ is a set of constraints introduced by resolution steps in Δ s.t. $(c, e \setminus c_1 ... \setminus c_n)$ is satisfiable*
$< c\|\alpha, \beta \ ; \ \Delta >$	*del−atoms(α)*	$< c\|\beta \ ; \ \Delta \ominus \alpha >$
$< c\|\beta \ ; \ \Delta >$	*add−atoms(α)*	$< c\|\alpha, \beta \ ; \ \Delta \oplus \alpha >$

Here the addition of a constraint is a complex operation which, in case of inconsistency, restores consistency by first removing a subset of unsatisfiable constraints from the derivation, and then by delaying the corresponding resolution step. In this operation, the choice of a precise subset of satisfiable constraints in the derivation is left unspecified. In our implementation this choice is based on the dependency informations used by the incremental constraint solver presented in the next section[2].

One can easily check that the transformations define correct CSLD derivations for the transformed goals.

Lemma 3.5 (Soundness of the transformations). Let Δ be a CSLD derivation for a goal G, and $< G' \ ; \ \Delta' >$ be the transformed goal and derivation obtained by some goal manipulation command. Then Δ' is a CSLD derivation for G'.

Example 3.6. The transformation for the addition of a constraint can be illustrated by the following typical disjunctive scheduling CLP program over natural numbers:

```
disj(X,Y,DX,DY):-Y >= X + DX.
```

[2]Note that in our context a notion of maximally satisfiable subset should take into account the dependencies between resolution steps in the derivation.

```
disj(X,Y,DX,DY):-X >= Y + DY.
```

Let us consider the following successful derivation:

$m \geq x, m \geq y, m \geq z \mid \underline{disj(y,z,2,1)}, disj(x,z,1,1), disj(x,y,1,2)$

$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2 \mid \underline{disj(x,z,1,1)}, disj(x,y,1,2)$

$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2, z \geq x+1 \mid \underline{disj(x,y,1,2)}$

$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2, z \geq x+1, y \geq x+1 \mid \square$

Now the addition of the constraint $2 \geq m$ *to the goal causes an inconsistency in the derivation, more precisely with the constraint* $z \geq y+2$ *introduced in the first resolution step.*

The effect of the command add_constraint$(2 \geq m)$ *is to restore consistency by removing the constraint introduced by the first resolution step. The transformed derivation is thus obtained simply by delaying the first resolution step:*

$2 \geq m, m \geq x, m \geq y, m \geq z \mid disj(y,z,2,1), \underline{disj(x,z,1,1)}, disj(x,y,1,2)$

$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2 \mid disj(y,z,2,1), \underline{disj(x,y,1,2)}$

$\rightarrow 2 \geq m, m \geq x, m \geq y, m \geq z, z \geq x+1, y \geq x+1 \mid \underline{disj(y,z,2,1)}.$

Then the search continues from that derivation, giving here a success immediately in one resolution step:

$\rightarrow 2 \geq m, m \geq x, m \geq y, m \geq z, z \geq x+1, y \geq x+1, y \geq z+1 \mid \square.$

Now the execution model consists of developing a new CSLD tree containing the transformed derivation for the modified goal. As any CSLD derivation for a goal G can be completed in a CSLD tree for G, the completeness of the execution model for reactive CLP trivially follows from lemma 3.5.

3.3. Requirements for the Constraint Solver

The operations on CSLD derivations described in the previous section suppose that the constraint solver can

1. add or delete a constraint from a system of constraints and check its satisfiability incrementally (operations `add(c)` and `del(c)`).
2. given a consistent system of constraints c and an atomic constraint a such that $a \cup c$ is inconsistent, identify a (minimal) subset $d \subseteq c$ such that $a \cup c \setminus d$ is consistent (cf. transformation *add-constraint(a)*).

The existence of an incremental constraint solver for checking the satisfiability of a stack of constraints is a standard assumption in CLP, but here we do not assume any longer that the system of constraints is a stack: any constraint can be deleted by a `del` operation, not necessarily the last added constraint as in a stack. We thus assume full incrementality of the constraint solver w.r.t. the set operations of addition and deletion.

The possibility to delete any constraint from the store imposes to carefully revise some optimizations of the constraint solver such as the removal of redundant constraints, as the deletion of a constraint can obviously change the status of a constraint from redundant to active. In CLP($\mathbf{R}$), the Simplex algorithm can be made

fully incremental as long as all constraints remain in the tableau. In CLP(FD), the situation is more complex as the reductions of domains appear as redundant constraints whose dependency need be handled appropriately. Some algorithms have already been proposed for dynamic constraint satisfaction problems [2] [11] [31]. In the next section we describe a generic fully incremental constraint propagation algorithm, and prove its completeness in a simple framework of constraints as closure operators.

The second requirement concerns the capability of the constraint solver to identify the causes of an inconsistency. This problem has been well studied in the context of intelligent backtracking for Prolog [7], CLP($\mathbf{R}$) [9] and CLP(FD) [5]. For CLP($\mathbf{R}$), it is shown in [9] that the Quasi-Dual algorithm of Lassez [20] provides in fact a minimal conflict. For CLP(FD), we show in the next section how the dependency graph provides also a solution to this requirement.

4. INCREMENTAL CONSTRAINT PROPAGATION ALGORITHMS

In this section, we present the dynamic constraint solver used in our implementation of CLP(FD) with reactive capabilities. The algorithm is presented in an abstract framework, and illustrated with examples over finite domains. The algorithm basically combines the standard constraint propagation algorithm of CLP(FD) [28] to a deduction maintenance method which keeps track of all dependencies needed for the deletion of constraints, in a way similar to Doyle's Truth Maintenance Systems [12].

For the sake of simplicity and generality, the correctness of the algorithm is proved in an abstract setting where constraints are identified with closure operators in a lattice of environments. This setting is presented first.

4.1. Constraint Propagation as Chaotic Iteration of Closure Operators.

An *environment* $E : \mathcal{V} \rightarrow 2^{\mathcal{D}}$ associates a domain of possible values to each variable. The set of environments forms a lattice structure, $(\mathcal{E}, \sqsubseteq)$, for the *information ordering* defined by $E \sqsubseteq E'$ if and only if $\forall x \in \mathcal{V}\ E(x) \supseteq E'(x)$. Note the duality between the information ordering and the domain inclusion ordering: the union (i.e. least upper bound) of information in two environments corresponds to the intersection of domains.

Now the semantics of an atomic constraint b can be defined as a closure operator over $\mathcal{E}$, noted $\overline{b}$, i.e. a mapping $\mathcal{E} \rightarrow \mathcal{E}$ satisfying

i) (extensivity) $E \sqsubseteq \overline{b}(E)$,

ii) (monotonicity) if $E \sqsubseteq E'$ then $\overline{b}(E) \sqsubseteq \overline{b}(E')$

iii) (idempotence) $\overline{b}(\overline{b}(E)) = \overline{b}(E)$.

In [26], this approach is developed in the lattice of constraint stores, and is generalized to the semantics of concurrent constraint programs. Here our purpose is different, we want to analyze constraint propagation algorithms in this abstract framework, in order to use the *abstract properties* of constraint propagation algorithms for giving a simple proof of correctness of the constraint retraction algorithm.

As is well known the union of closure operators is not a closure operator, the semantics of a system of constraints c is thus not simply the union of the closure

operators of the atomic constraints in c, but the closure operator

$$\overline{c} = fix(\lambda E.E \sqcup \bigsqcup_{b \in c} \overline{b}(E)).$$

Example 4.1. Let us consider the inconsistent system of constraints in CLP(FD) composed of two constraints, $b : x > y$ and $c : y > x$.

In an initial environment $E(x) = [1, 10]$, $E(y) = [1, 10]$, we have $\overline{b}E(x) = [2, 10]$, $\overline{c}E(x) = [1, 9]$, thus $(\overline{b} \sqcup \overline{c})E(x) = [2, 9]$, whereas the closure operator $\overline{b, c}$ associated to the conjunction of constraints b, c gives the intended semantics of the conjunction of constraints: $\overline{b, c}E(x) = \emptyset$.

Now the method of approximating $\overline{c}$ by iterating the closure operators associated to the atomic constraints in c is faithful to constraint propagation algorithms for solving systems of constraints (note however that termination is not assumed at this stage). This method is a particular case of the very general chaotic iteration method for solving a system of fixed point equations in a lattice [8].

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, n an integer strictly positive, and $F : L^n \to L^n$ a monotone operator over L^n, where $F_i : L^n \to L$ denotes the projection of the function on its ith component. The *chaotic iteration* of F from $D \in L^n$ for a transfinite choice sequence[3] $< J^\delta : \delta \in Ord >$ of parts of $\{1, ..., n\}$ satisfying $\{\forall \delta \in Ord, \forall i \in \{1, ..., n\}, \exists \alpha \geq \delta : i \in J^\alpha\}$, is the transfinite sequence $< X^\delta : \delta \in card(L) >$ of elements in L^n defined by:

$X^0 = D$,

${X_i}^{\delta+1} = F_i(X^\delta)$ if $i \in J^\delta$, ${X_i}^{\delta+1} = {X_i}^\delta$ otherwise,

${X_i}^\delta = \bigsqcup_{\alpha<\delta} {X_i}^\alpha$ for any limit ordinal δ.

Theorem 4.2 ([8]). Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, n an integer strictly positive, $F : L^n \to L^n$ a monotone operator over L^n, and $D \in L^n$ a pre fixpoint of F (i.e. $D \sqsubseteq F(D)$). Then any chaotic iteration of F starting from D is increasing and has for limit the least fixed point of F greater than D.

Corollary 4.3 (Correctness of constraint propagation). Let c be a system of atomic constraints $a_1, ..., a_n$. Let E be an environment. Then $\overline{c}(E)$ is the limit of any fair iteration of closure operators $\overline{a}_1, ..., \overline{a}_n$ from E.

PROOF:

Consider the following system of $n + 1$ fixed point equations, and the function $F : L^{n+1} \to L^{n+1}$ defined by its projections, $F_1, ...F_{n+1}$:

$$\begin{cases} E_1 = \overline{a}_1(E) = F_1(E_1, \ldots, E_n, E) \\ E_2 = \overline{a}_2(E) = F_2(E_1, \ldots, E_n, E) \\ \ldots \\ E_n = \overline{a}_n(E) = F_n(E_1, \ldots, E_n, E) \\ E = E_1 \cap \cdots \cap E_n = F_{n+1}(E_1, \ldots, E_n, E) \end{cases}$$

[3] In our use of chaotic iteration for modeling constraint propagation, transfinite sequences need not be considered as termination is ensured in our case, nevertheless we express the soundness theorem in its general form.

The functions F_i's are obviously monotonic, any fair iteration of closure operators $\overline{a}_1, ..., \overline{a}_n$ is thus a chaotic iteration of $F_1, ..., F_{n+1}$ (with F_{n+1} applied in the even indices of the sequence[4]), therefore its limit is equal to the least fixed point greater than E: $\overline{c}(E)$. □

Note that in [1], a more general theorem is proved where the idempotency assumption of the closure operators is relaxed. In [26], a similar approach is used for the semantic foundation of concurrent constraint programming, where the denotation of an agent is a closure operator in the (dual) lattice of constraint stores.

4.2. Incremental Constraint Propagation Algorithm with addition and deletion of constraints

According to the previous section, incremental constraint propagation can be modeled by a transition system $(\Gamma, \Longrightarrow)$ where Γ is a set of configurations and $\Longrightarrow \subseteq \Gamma \times \mathcal{C} \times \Gamma$ is the transition relation labeled by atomic constraints.

For dealing with deletion of constraints, information about dependencies must be maintained. Consequently a configuration will be a triple

$$< E : \mathcal{V} \to 2^{\mathcal{D}}, P : \mathcal{C} \times \mathcal{V} \to 2^{\mathcal{D}}, C : \mathcal{V} \to 2^{\mathcal{C}} >$$

composed of an environment E, a *producer*[5] function P which associates with an atomic constraint and a variable, the set of domain values removed by the constraint from the domain of that variable, and a *consumer* function C which associates with a variable the set of atomic constraints which have used the domain of the variable to reduce other domains.

Definition 4.4. Let $\mathcal{C}$ be a system of constraints $a_1, \ldots, a_n$. The transition relation for constraint propagation is defined as the least relation over configurations $\Longrightarrow$ satisfying the following rules:

$$Fail \quad \frac{a_i \in \mathcal{C} \qquad x \in \mathcal{V}(a_i) \qquad \overline{a_i}E(x) = \emptyset}{< E, P, C > \stackrel{a_i}{\Longrightarrow} fail}$$

$$CP \quad \frac{a_i \in \mathcal{C} \qquad E' = \overline{a_i}(E) \qquad E' \neq E \qquad E' \neq \emptyset}{< E, P, C > \stackrel{a_i}{\Longrightarrow} < E', P', C' >}$$

where $P'(a_i, x) = P(a_i, x) \cup (E(x) \setminus E'(x))$ for all $x \in V(a_i)$,
$P'(a_i, x) = P(a_i, x)$ for all $x \notin V(a_i)$,
$C'(x) = C(x) \cup \{a_i\}$ for every $x \in \mathcal{V}(a_i)$ such that $E'(y) \neq E(y)$ for some $y \in \mathcal{V}(a_i) \setminus \{x\}$,
$C'(x) = C(x)$ otherwise.

[4] One can remark in the proof that the propagation of atomic constraints need not be synchronized, constraint propagation can be done in parallel, provided that the synchronization equation $F_n + 1$ is not discarded indefinitely.

[5] The term producer originates from the Herbrand constraint system where the effect of equality constraints is to produce substitutions, whereas in FD the effect of constraints is to produce domain restrictions.

As a consequence of proposition 4.3, we get that for a constraint system c, any terminating sequence of transitions labeled by atomic constraints in c, starting from an initial environment E, ends with final environment $\overline{c}(E)$. Hence the operation of addition of a constraint, `add(c)`, can simply add c to the system of constraints and apply the transitions up to a fixed point.

But the maintenance of producers and consumers gives much more and allows to implement the other operation of deletion of constraints `del(c)`. That operation deletes the constraint c from the system of constraints, restores a consistent environment obtained by an operation of constraint relaxation called *relax(c)*, and then applies the transitions up to a fixed point. In order to define the operation *relax(c)*, let us remark that the labeled transition system defines a complex dependency graph between constraints variables and values. A simpler graph was used for intelligent backtracking in [5]. In that case, only the constraints responsible for the unsatisfiability needed to be determined, so only the producer function was required. Here for the operation of constraint relaxation we shall make use of a different subgraph which traces the effects of constraint propagation:

Definition 4.5. The constraint dependency graph in a configuration $< E, P, C >$ *is the graph of atomic constraints such that there is an arc from* a *to* a' *if and only if there exists a variable* $x \in \mathcal{V}(a)$ *such that* $P(a, x) \neq \emptyset$ *and* $a' \in C(x)$.

This graph is a subgraph of the graph of constraints containing an arc between each pair of constraints having a variable in common. Informally, there is an arc from a to a' in the constraint dependency graph, if a has reduced the domain of some variable checked by the constraint a'. Note that the constraint dependency graph is not optimal in the sense that it forgets which constraints removed which value from a variable domain. For practical efficiency reasons we chose to consider only the dependency information between constraints.

Definition 4.6. The operation of constraint relaxation is defined formally as a transformation of both the constraint system c *and the configuration* $< E, P, C >$ *by*

$[c, < E,\ P,\ C >]$ *relax(a)* $[c \setminus a, < E',\ P',\ C' >]$

where $E'(x) = E(x) \cup \bigcup_{a' \in S} P(a', x)$,
$P'(a', x) = \emptyset$ *if* $a' \in S$, *otherwise* $P'(a', x) = P(a', x)$,
$C'(x) = C(x) \setminus S$ *if* $x \in \mathcal{V}(S)$, *otherwise* $C'(x) = C(x)$,
S *is the set of constraints* a' *such that there exists a path from* a *to* a' *in the constraint dependency graph of* $< E, P, C >$.

One difficulty in proving the correctness of dynamic constraint solvers is that the environment obtained by constraint relaxation may be not reachable by constraint propagation. An immediate consequence of the fact that a constraint is a closure operator gives however a simple correctness criterion:

Proposition 4.7. Let c *be a constraint. If* $E \sqsubseteq E' \sqsubseteq \overline{c}(E)$, *then* $\overline{c}(E) = \overline{c}(E')$.

Soundness of constraint relaxation can thus be established simply by showing that the environment obtained by the relaxation of some atomic constraint lies between the initial environment and its fixed point for the constraint system without the deleted atomic constraint.

Lemma 4.8. Let $< E_0,\ P_0,\ C_0 > \stackrel{a_1}{\Longrightarrow} \ ... \stackrel{a_n}{\Longrightarrow} < E_n,\ P_n,\ C_n >$ *be a finite transition sequence with constraint system* c. *Let* $[c,\ < E_n,\ P_n,\ C_n >]$ *relax(a)* $[c \setminus a,\ < E,\ P,\ C >]$ *be the configuration obtained after the deletion of atomic constraint* a. *Then* $E_0 \sqsubseteq E \sqsubseteq \overline{c \setminus a}(E_0)$.

PROOF: By induction on n and a simple case analysis. □

Proposition 4.7 and lemma 4.8 together show the soundness of the incremental constraint solver with addition and deletion of constraints.

Theorem 4.9. Let E *be an initial environment and* c *an initial constraint system. Let* E' *be an environment obtained by the incremental constraint solver from* E *and* c, *by applying a sequence of addition and deletions of constraints resulting in a constraint system* d. *Then* $E' = \overline{d}(E)$.

PROOF: By recurrence on the length of the sequence of operations. □

It is worth noting that, unless the process of constraint propagation itself can be interrupted by constraint deletion commands, in the use of theorem 4.9 the initial environment E is a fixed point of $\overline{c}$. In this case not all atomic constraints in $d = c \setminus a$ need be propagated after the relaxation of a. The information contained in the producers and consumers of the configuration allows to determine the subset of constraints in d which have to be repropagated.

Example 4.10. Let us consider the following system of constraints over integers: $y \geq z + 1$, $y \geq x + 1$ *and* $t \geq z + 1$ *noted* a_1, a_2 *and* a_3 *respectively.*

Let $< E_0,\ P_0,\ C_0 >$ *be the initial configuration with* $E_0(x) = [1, 10]$, $E_0(y) = [2, 10]$, $E_0(z) = [1, 10]$, $E_0(t) = [1, 9]$, $P_0 = \emptyset$ *and* $C_0 = \emptyset$.

We have a transition sequence $< E_0,\ P_0,\ C_0 > \stackrel{a_1,a_2,a_3}{\Longrightarrow} < E_3,\ P_3,\ C_3 >$ *where* $E_3(x) = [1, 9]$, $E_3(y) = [2, 10]$, $E_3(z) = [1, 8]$, $E_3(t) = [2, 9]$

$P_3(a_2,\ x) = \{10\}$, $P_3(a_1,\ z) = \{10\}$, $P_3(a_3,\ z) = \{9\}$, $P_3(a_3,\ t) = \{1\}$

$C_3(y) = \{a_1, a_2\}$, $C_3(z) = \{a_3\}$, $C_3(t) = \{a_3\}$.

Now if the constraint a_1 *is deleted we obtain by constraint relaxation the state:*

$[c,\ < E_3,\ P_3,\ C_3 >]$ *relax(*a_1*)* $[c \setminus a_1,\ < E,\ P,\ C >]$
where $E(x) = [1, 9]$, $E(y) = [2, 10]$, $E(z) = [1, 10]$, $E(t) = [1, 9]$, $P(a_2,\ x) = \{10\}$ *and* $C(y) = \{a_2\}$.

From this state, constraint propagation terminates :with a single transition

$< E,\ P,\ C > \stackrel{a_3}{\Longrightarrow} < E_4,\ P_4,\ C_4 >$ *with*

$E_4(x) = [1, 9]$, $E_4(y) = [2, 10]$, $E_4(z) = [1, 8]$, $E_4(t) = [2, 9]$

$P_4(a_2,\ x) = \{10\}$, $P_4(a_3,\ z) = \{9, 10\}$, $P_4(a_3,\ t) = \{1\}$

$C_4(y) = \{a_2\}$, $C_4(z) = \{a_3\}$, $C_4(t) = \{a_3\}$.

Note that similar algorithms have been proposed in the literature on dynamic constraint solving [2] [11] [31]. In particular [2] proposes a similar trade-off between precision and efficiency of constraint relaxation, obtained by not tracing the removed values, but only the variables touched by a constraint.

4.3. Causes of inconsistency

The second requirement of the constraint solver is, given a consistent system of constraints c and an atomic constraint a such that (c, a) is inconsistent, to choose a subset $d \subseteq c$ such that $(a, c) \setminus d$ is consistent.

We have adopted a simple strategy based on the information contained in the producers. The system simply adds and propagates constraint a and puts repeatedly in d any constraint b such that $P(b, x) \neq \emptyset$ for some variable x in $\mathcal{V}(a)$, until a becomes consistent with $c \setminus d$.

Of course, this strategy still leaves some choices unspecified and does not compute an optimal subset, but it does have the effect of localizing the conflicts to a subset of constraints easily determined by the dependency information on constraint propagation. On the other hand it is worth noting that because of the dependencies between atoms, computing an optimal subset of constraints, that is a satisfiable subset of constraints of maximal cardinality, does not necessarily lead to a minimal revision of the derivation. Therefore the right notion of optimality is rather complex and although of theoretical interest, seems hardly amenable to efficient implementation.

5. IMPLEMENTATION

Our implementation of CLP(FD) called Meta(F) [21] [5] is based on SICStus Prolog [27]. The constraint solvers are written in C and interfaced with SICStus Prolog through the standard interface. The performances of Meta(F) are comparable to the state-of-the-art implementations of CLP(FD) (typically 7 times as fast as the previous version written completely in Prolog reported in [5], or the CLP(FD) library in [27]).

In the reactive version of Meta(F), the constraint dependencies defined in the previous section are fully managed by the constraint solver in C. This is responsible for a time overhead of less than 15% w.r.t. constraint propagation in the standard version of Meta(F).

On the other hand, the dependencies among atoms and the reactive search procedure are managed by a meta-interpreter. For these reasons the initial overhead of the reactive version w.r.t. the standard version can be more important depending on the trade-off between backtracking and constraint propagation. Our experiments with scheduling problems showed that the time overhead ranged from 10% to twofold in some proofs of optimality. In the applications reported in the following sections, we shall see that this overhead is acceptable and that the benefit of incrementality pays off in these applications where the speed-up can attain two orders of magnitude. The next section describes the reactive search procedure.

5.1. Reactive search procedure

We recall that a derivation of a CSLD tree is transformed by a goal manipulation command to give a new derivation, which is the point of departure for the development of a CSLD tree for the new goal (see figure 1). A reactive search procedure has to explore an entire CSLD tree from an internal node, that is from the transformed (partial) derivation resulting from the goal manipulation command. The subtree below the internal node is searched first, then the other portions of the CSLD tree are searched by remounting the derivation to the root.

The incremental constraint solver with addition and deletion of constraints makes it possible to implement such a reactive search procedure with a simple meta-interpreter.

5.1.1. Simulating Backtracking with Iterative Search. In order to present the reactive search procedure we first illustrate the flexibility acquired by the presence of an explicit operation for removing a constraint from the store (operation `del(c)`). The iterative search procedure traverses a CSLD tree in a depth first, left to right order, simulating backtracking by `add(c)` and `del(c)` operations.

For the sake of simplicity, the following meta-interpreter assumes (without loss of generality) that each predicate is either undefined or defined by exactly two clauses. The predicate `clauses(A,[C1|B1],[C2|B2])` states that the atom A is defined by the rules $A \leftarrow c_1 | \beta_1$ and $A \leftarrow c_2 | \beta_2$ where β_1 and β_2 are represented by lists of atoms `B1`, `B2`.

```
search([]) :- success.
search([A|G]) :-
       (   clauses(A,[C1|B1],[C2|B2]) ->
            (   add(C1) ->
                append(B1,G,G1),
                search(G1),
                del(C1)
            ;   true
            ),
            (   add(C2) ->
                append(B2,G,G2),
                search(G2),
                del(C2)
            ; true
            ),
        ; true
        ).
```

The meta-interpreter is called with the predicate `search`. The argument is a goal given as a list of atoms. If the derivation is successful the predicate `success` is called, otherwise the search continues through `del` and `add` operations.

5.1.2. Reactive Search for Dynamic CLP Problems. The reactive search procedure takes into account interactions and combines the iterative search procedure with the derivation transformations presented in section 3.

The reactive search meta-interpreter given in table 2 is called by the predicate `search` with one argument: the goal given as a list of atoms. The predicate `search` with two arguments keeps track of the derivation in its second argument . The derivation is represented as a list of tuples, formed with the clause body used in each derivation step together with the alternative clause body and the father goal. The predicate `querymodification` takes into account the interactions. The predicate `transformation` transforms the goal and the derivation as described in section 3.

After a modification of the query, `react` first searches for a successful derivation from the current derivation using the predicate `search`. If a successful derivation is found then the predicate `success` is called to signal the success and wait for interactions.

```
search(G) :- search(G,[]).

search([],Der) :- success(Der).
search([A|G],Der) :-
        (   clauses(A,[C1|B1],[C2|B2]) ->
            (   add(C1) ->
                append(B1,G,G1),
                react(G1,[(choice1(C1,B1),choice2(C2,B2),goals(G))|Der]),
                del(C1)
            ;   true
            ),
            (   add(C2) ->
                append(B2,G,G2),
                react(G2,[(choice1(C2,B2),choice2(C1,B1),goals(G))|Der]),
                del(C2)
            ;   true
            )
        ;   true
        ).

react(G,D) :-
        (   querymodification(G,D,M) ->
            transformation(M,G,D,G1,D1),
            search(G1,D1),
            backsearch(D1)
        ;   search(G,D)
        ).

backsearch([]) :- abort.
backsearch([(choice1(C1,B1),choice2(C2,B2),goals(G))|Der]) :-
        del(C1),
        (   add(C2) ->
            append(B2,G,G2),
            search(G2,[(choice1(C2,B2),choice2(C1,B1),goals(G))|Der]),
            del(C2)
         ;  true
         ),
         backsearch(Der).
```

Table 2. Meta-interpreter for Reactive Search

After the search from the transformed derivation is exhausted, the reactive search procedure to be complete has to explore the rest of the CSLD derivation tree. The predicate `backsearch` explores the other portions of the CSLD tree by remounting the derivation.

Note that in this version, at any point the search process may be interrupted and the goal modified, starting a new search from the transformed derivation. Note also that for the sake of simplicity, this meta-interpreter keeps continuations which may be abandoned. After `backsearch` is exhausted the process aborts, with no need to execute the continuations attached before the last query manipulation. This defect can be fixed however to fit hypothesis 2 on the independence of the space complexity on the number of interactions. In the applications described in the following sections, the dependencies between atoms are in fact handled in an ad hoc fashion for efficiency reasons w.r.t. both memory and time.

6. EVALUATION ON A MULTI-FREQUENCY ALLOCATION PROBLEM

We have applied the reactive CLP scheme to the allocation of frequency bands from a radio spectrum to a group of networks. The radio spectrum has two to three thousand distinct frequencies. The system allocates frequency bands for several hundred networks, by partitioning them into strongly connected subsets containing about twenty networks, and allocating frequencies to these subsets. The allocation of frequencies to networks is constrained to respect forbidden frequency constraints, network capacity constraints, and interference constraints that guarantee that when two networks are close, the bands of frequencies that they are allocated will be sufficiently distant to avoid interference. The separation of bands of frequencies for two networks is a function of the degree of proximity and of the frequencies allocated. The higher the frequency allocated the greater the separation.

A typical allocation is presented in the figure 2. In this image the spectrum associated with a network is represented in light grey. The spectrum corresponds to the range of all the frequencies available for allocation to the networks. The forbidden frequencies of the spectrum and the frequencies allocated to the network are illustrated by the dark blocks and the white blocks respectively.

The major difficulty of the problem lies in the definition of what constitutes a good placement for the bands of frequencies because several criteria contribute to the quality of a good frequency allocation. Let us examine the following criteria for improving the resistance to interference and interception: the maximization of the number of frequencies allocated per network, and the maximization of the separation between the two bands on the same network. Consider for example two networks i and j which are close. If the separation between the two bands of network i is increased, the frequencies available to network j are reduced. This in turn reduces the possibility of maximizing the number of frequencies for the network j. In addition to the multi-criteria optimization techniques described in [13] the capability of the system to react to the interactions of the user to skip from one solution to another is of prime importance. The interactions are the composition of several goal manipulations. The operator may interact with the problem in any one of the following ways :

1. Increase the number of frequencies `NFj` of a selected network `j` (the interaction "better freq" in figure 2). The basic idea is to add the constraint, `add_constraint(NFj > cj)`, to the goal stating that the number of allocated frequencies to the selected network must be gretaer than the number

Figure 2. A sample of multi-frequency allocation screen dump

available in the present allocation. This constraint is not compatible with the current derivation and thus the transformation of the derivation solicits the `del` operation to remove a subset of constraints that are unsatisfiable with the introduction of this constraint to the solver. The choice of such a subset is controlled by first adding compatible constraints `add_constraint NFi>=ci` to force the number of frequencies allocated to each of the networks after the interaction will be greater than or equal to the number of frequencies allocated before the interaction. Similar constraints are added for the separation of frequencies.

The meta-interpreter then updates the derivation and the reactive search procedure begins its exploration for a new successful derivation with the hard constraint of not decreasing the quality of the allocation to the other networks. The frequency allocation of some networks is then modified, in fact the modified networks correspond to the constraints removed to "cure" the unsatisfiability of the previous derivation.
An example of selective optimization is shown in figure 2. The user chooses to increase the separation between the frequency bands of the network 13. The modifications with respect to the preceding solution are colored with a darker grey (cf figure 3). Notice that network 13 is not the only network that has changed. The networks 14, 15, 16, 17 have also been modified. This is explained by the fact that these different networks are related by non-interference constraints. All the networks have a separation between their bands of frequencies which is better or equal than before. It happens that the 15th network has also an increase in the number of frequencies allocated. This result is obtained by the constraints that are added during the interaction which impose that the solution is not degraded.

2. Increase the separation of the two bands of a selected network `j` the interac-

Figure 3. Modified solution after asking to increase separation in network 13.

tion "better sep". This is similar to the interaction "better freq".

3. Modify the forbidden frequency bands. In this problem, the forbidden frequency bands change over time, certain become available while others become forbidden. This interaction deletes the constraints in the goal (by `del_constraint`) due to the modified forbidden frequencies and adds the new forbidden frequencies constraints to all networks (`add_constraint`). The evolution over time of a problem and its description in terms of goal manipulation commands is described in greater detail in the following application.

In a typical scenario an initial allocation is found automatically with some heuristics and a fixed optimization criterion. The solution is then progressively improved by repeatedly selecting a network and improving the number of frequencies it has been allocated, or by increasing the separation of its frequency bands. The operational model in effect has to perturb a solution in such a way that the solution found after an interaction improves a criterion, without departing too much from the solution before the interaction.

The incremental search strategy is well-suited to this application because the dynamic dependency graph of constraints contains several connected components. The effect of a goal manipulation command is thus localized to a subgraph of constraints and resolution steps. A new solution is found incrementally by revising a subset of choices for the previous solution. Only some parameters of the previous solution are thus revised and the incremental search strategy allows to converge towards better solutions in an interactive manner.

7. EVALUATION ON AN AIRCRAFT SEQUENCING PROBLEM

The time speed-up obtained with our incremental execution model has been evaluated in another application of dynamic rescheduling in air traffic control. The terminal zone aircraft sequencing problem (ASP) represents an important bottleneck of air traffic flow management. Its statement can be summarized as follows: given a set of aircraft entering in the terminal area (e.g. 30 min from the airport) determine an optimal sequence, according to terminal configurations, procedural safety constraints, aircraft capacities and expected schedule. The usual optimal criteria is the completion time of the sequence. In addition dynamic aspects of the problem must be accounted for, such as: the arrival of new flights in the terminal area, the temporal evolution of the problem and unexpected events such as the rerouting of flights.

The optimal resolution of this task using manual techniques has become impossible because of increase in the number of flights. In fact peek traffic can be as high as a flight a minute. The strategy adopted in most airports is that of first come first served (FCFS). This strategy is easy to implement but can create delays. Our goal is to find a sequencing strategy capable of producing better sequences and maintaining them while taking into account the interactions of the environment with the system.

7.1. Static scheduling

Figure 4 shows the graphical user interface (GUI) of the application that sequences flights in the terminal zone. In the figure, the flight sequence shown in the lower part of the GUI is obtained using the FCFS strategy. The flights are represented by circles. The sequence shown in the upper part of the GUI is found using our sequencing program. The lower and upper sequences are different. The FCFS strategy sequences flights in the order 1..20. The system finds an optimal solution which in this example simply inverts the order of flights 19 and 20.

Not all sequences of flights are possible because aircrafts must respect procedural constraints in the terminal zone. The terminal zone is composed of three zones: the critical zone, the regulated zone and the non-regulated zone. In each part of the terminal zone corresponds a flight time from the zone to the runway. In this image the zones are from left to right: the critical zone in dark grey (5min from runway), the regulated zone in light grey (5min to 20min from runway), and the non-regulated zone in dark grey (20min to 35min from runway).

- Each flight must follow a pre-established route, an air corridor. The air corridors are separated by the white lines in figure 4. For example flights 10 and 12 are in different air corridors.

- Each flight has a predicted arrival time. The predicted arrival time determines the zone in which the flight is located. For example flight 1 has a predicted arrival time which is less than 5 minutes and therefore is in the critical zone. In the figure 4 the zone in which the flight is affected can be seen in the lower half of the screen.

- The scheduled arrival time. The arrival time of each flight is limited around the predicted arrival time by an advance factor and a delay factor. In the following, we call arrival time the scheduled arrival time, not to be confused with predicted arrival time.

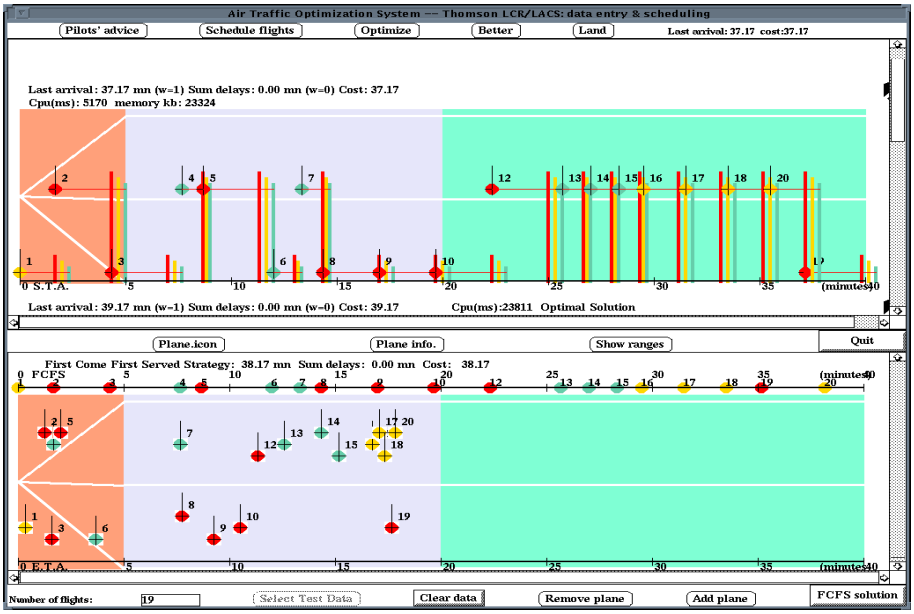

Figure 4. A sample of aircraft sequencing solutions.

- The order of flights in the critical zone is fixed. For example flights 1 and 2 are in the critical zone. Flight 1 has an earlier predicted arrival time than that of flight 2. So flight 1 always preeceds flight 2.
- The order of flights in the regulated zone and in the same air corridor is fixed. For example flights 7 and 12 are in the same air corridor. Flight 7 has an earlier predicted arrival time than that of flight 12. So flight 7 always preeceds flight 12. Flights 7 and 8, however, can be sequenced in the order 7,8 or 8,7.
- A safety time must be respected between the arrivals of flights. The time to be respected is a function of the couple class of flight on the runway and the class if next flight to land. For example if flight 13 follows flight 12 it must be at least 3 minutes later but if flight 12 follows flight 13 it need only be 1 minute later.

Finally the objective is to minimize the completion time of the sequence.

The problem is modeled as a disjunctive scheduling problem with variable duration tasks. The starting date of the tasks are represented by the Scheduled Time of Arrival variables `STAx` for each flight x.

The procedural constraints and objective function are modeled as follows.

- Let the advance factor and the delay factor around the predicted arrival time (`pt`) be `af` and `df` respectively.

$$\forall x max(0, pt - af) \leq STAx \leq pt + df$$

- Let flights x and y be ordered such that x is before y.

  `STAx < STAy`

- Flights x and y must respect the safety distance of Dxy and Dyx respectively. The following clauses introduce a disjunction which establishes the fact that the flight x and y can not take place at the same time.

```
disj(STAx,STAy,Dxy,Dyx) :- STAx + Dxy <= STAy
disj(STAx,STAy,Dxy,Dyx) :- STAy + Dyx <= STAx
```

  For efficiency reasons however a constructive disjunction constraint [18] [30] is in fact used here.

- Finally the objective function `Cost` can be modeled as a variable which is greater than the scheduled arrival time of all variables.

```
Cost <= STAx
```

7.2. Dynamic rescheduling

The interactions supported by the system are described in terms of goal manipulation operations.

- The addition of a flight. This interaction creates a new variable for the scheduled time of arrival of the new flight, and requires the addition of precedence constraints with other non permutable flights with `add_constraint` operations. It also requires the addition of a disjunctive atom

```
add_atom(disj(STAx,STAy,Dxy,Dxy))
```

  to the goal for each flight in the terminal zone which can be permuted with the added flight.

- The removal of a flight consists of removing the constraints (`del_constraint`) and atoms (`del_atom`) introduced by the presence of the flight.

- The landing of a flight. This interaction requires a careful management of the temporal evolution of the problem. All the flight plans of the airplanes must be updated by the time it takes the first flight in the sequence to land. This is translated as `add_constraint` commands, because the safety distances to be respected by the flights are increasingly severe as they approach the runway.

- The optimization of the landing sequence. Optimization by branch and bound can be modeled in this framework as a particular case of interaction with an agent which repeatedly constraints the final scheduled time of arrival of the sequence (`add_constraint`).

7.3. Performance results

Table 3 presents the computation times for finding a solution to the sequencing problem subject to typical interactions. The interactions on the problem are the following. The initial problem is to schedule 20 flights in two corridors (pb 1), then the first flight lands (pb 2), a new flight enters the terminal zone (pb 3), a second flight enters the terminal zone (pb 4), a flight is rerouted (pb 5), a second flight lands (pb 6), a third flight arrives in the terminal zone (pb 7), a third flight lands (pb 8), and finally a fourth flight arrives in the terminal zone (pb 9).

pb	S	R	S_{opt}	R_{opt}	S/R	S_{opt}/R_{opt}
1	99	110	2909	3257	**0.90**	**0.89**
2	160	10	3448	40	**16.00**	**86.20**
3	149	20	3839	10	**7.45**	**384.00**
4	160	10	10050	88	**16.00**	**114.00**
5	140	1510	3949	4380	**0.09**	**0.90**
6	129	20	3907	60	**6.45**	**65.10**
7	99	20	9425	528	**4.95**	**17.90**
8	99	10	8050	60	**12.00**	**134.00**
9	99	10	5179	10	**14.00**	**518.00**

Table 3. Computation times ratio between static and reactive resolution.

The times given for S and R represent the computation time required to find a solution for the problem using *static resolution by reexecution* and *reactive incremental resolution* respectively. Times are also presented for finding an optimal sequence after each evolution of the problem w.r.t. the static approach (S_{opt}) and the reactive one (R_{opt}). The standard version Meta(F) was used for the static problem solving and its reactive experimental extension for the reactive resolution. Note that an intermediate model of execution using oracles based on the previous solution and reexecution could be evaluated with the static resolution. This has not been experienced in this application.

The evaluation shows that the operational model we propose for constraint logic programming is efficient w.r.t. the normal execution model. It achieves a speed up of one order of magnitude for non-optimized landing sequences and of two orders of magnitude for optimized landing sequences. The exception to this rule being the fifth interaction, that is the rerouting of an airplane. The slow down for this interaction is caused by the current implementation of the deletion of atomic constraints and atoms one by one instead of in one single operation.

Here again, the application is well-suited to the incremental search strategy because the dynamic dependency graph of constraints contains several connected components. Solutions can be found by permuting flights incrementally without changing the other flights. This is detected automatically by the incremental constraint solver and exploited in the derivation transformation system of the reactive search procedure.

8. COMPARISON WITH OTHER WORK

The CLP reactive execution model is a very general model of execution and it is possible to compare it to methods based on re-execution with oracles [29], and to Maher and Stuckey's method [22] for the addition of constraints and atoms to the goal. In this section, we present these methods in terms of operations on derivations and compare them.

8.1. Re-execution with oracles

Van Hentenryck [29] proposes a method for re-executing a goal with an oracle, after the addition or removal of constraints in the goal. The oracle is used to develop a

CSLD tree for the modified goal and to explore first of all the branch of the CSLD tree described by the oracle. It is possible to analyze this model very simply in terms of operations on derivations.

Definition 8.1. An oracle for a goal G and a program P, noted Δ*, is a successful derivation for P,G.*

The transformation used to add a constraint to a goal is expressed in terms of the operation of pruning the derivation of the oracle.

Definition 8.2. Let Δ *a successful oracle for the goal* $d \mid \alpha$ *and the program P. The transformation for the addition of a constraint c is defined by the command:*

$$< d \mid \alpha, \Delta > oracle - add - constraint < c, d \mid \alpha, \Delta \otimes c > .$$

As a consequence all the information in the derivation after the first resolution step which is unsatisfiable with the added constraint is lost. In our model the addition of a constraint allows to use more of the resolution steps in the oracle because the unsatisfiable resolution steps are delayed.

In [29], two proposals are made for the transformation for the removal of a constraint. The first uses an oracle for a less constrained goal stocked in the system, to which the supplementary constraints are added with the pruning operator. The default of this method is that a large search space can be explored before the previous solution is recovered, although it clearly provides a solution to the less constrained query. The second method is based on the oracle given by the previous derivation obtained before the removal of the constraint. This operation is formalized by the lifting operation.

Definition 8.3. Let Δ *a successful oracle for a goal* $c, d \mid \alpha$ *and a program P. The transformation for the removal of the constraint c with heuristic is defined by the command:*

$$< c, d \mid \alpha, \Delta > oracle - del - constraint - heuristic < d \mid \alpha, \Delta \oslash c > .$$

With this method the lifted solution is immediately found. According to our search procedure however, all the alternatives to the choice points of the lifted derivation $\Delta \oslash c$ are explored by remounting the derivation (by predicate `backsearch`). In [29], an optimization is proposed based on a combination with the first method: the alternatives of the first choice points from the root are not explored if they have been pruned by an oracle for a less constrained goal (first method), then when the choice points of the preceding derivation differ from the oracle, the previous derivation is used only as a heuristic and all the other alternatives at the choice points are explored, like in our search procedure. For example, when the search is from left to right, all the derivations at the left of the first successful derivation are finitely failed, consequently these alternatives can be discarded for a less constrained goal. These optimizations can be integrated in our reactive search procedure with similar restrictions on the class of CSLD trees considered. They were not implemented however in the system that served to the evaluation.

8.2. The Maher-Stuckey Method

Maher and Stuckey [22] propose an execution model based on query manipulation commands. The deletion of constraints or atoms in the query is not considered, only addition and search of optimal solutions.

The Maher-Stuckey method is applicable to CSLD trees formed with the left to right selection strategy of Prolog. The addition of an atom to a goal $c|\alpha,\beta$ can be made between α and β. The transformation of the CSLD tree consists of cutting the derivations at the point which corresponds to the goal $c|\alpha$ and to insert a derivation for A and to graft the derivation tree for β. The figure 5 illustrates this transformation.

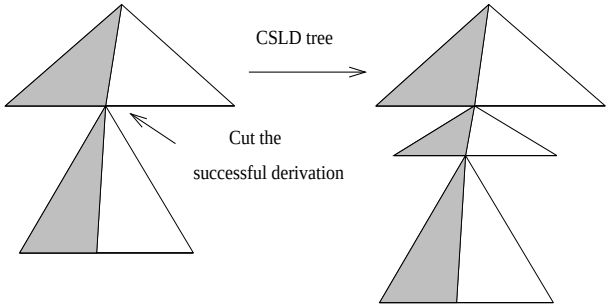

Figure 5. CSLD tree transformation and search procedure

The search procedure for this method is based on the exploration of the transformed tree. The procedure can be optimized to avoid the exploration of finitely failed derivations that have already been discovered. The gray part of the CSLD tree in figure 5 corresponds to the part of the CSLD tree that it is not necessary to re-explore.

Our execution model does not privilege a particular selection strategy, the operation of the addition of an atom in the middle of the goal does not affect the transformation which will always add the atom to the end of the derivation. The operation of atom addition in the Maher-Stuckey method thus corresponds to another derivation transformation that preserves the selection strategy and that can be formalized as follows:

Definition 8.4. Let $G = c|\alpha,\beta$ be a goal, and A an atom to add to the goal between α and β. Let Δ be a finite derivation for G of the form $\Delta = \Delta' \longrightarrow (d|\beta) \longrightarrow \Delta''$. Let Δ_1 be a derivation for $d|A$. The transformed derivation for the goal $c|\alpha, A, \beta$ is the derivation:

$\Delta' \longrightarrow (\Delta_1 \oplus \beta) \longrightarrow (\Delta'' \otimes e)$ *if Δ_1 is a successful derivation with computed answer e,*
$\Delta' \longrightarrow (\Delta_1 \oplus \beta)$ *if Δ_1 is a failed derivation for A.*

8.3. Discussion

Neither of the two methods above proposes an execution model for the complete set of goal manipulation commands. This reduces the interactions that it is possible to use on an application with these models. Consider the case of a scheduling problem

with disjunctive constraints, such as the aircraft sequencing problem (cf section 7). For this class of problems, we associate a variable with a task. Through a goal manipulation command we can add or remove a constraint to advance or delay the starting date of a task. Of course other interactions are possible with these manipulations, for example : the minimization or maximization of cost functions on solutions. If we wish to add a new task to the problem we must be able to add atoms to the problem to add disjunctive constraints to the goal. Similarly the removal of tasks from the problem necessitates to remove atoms from the query.

The transformations for the addition of the constraint mark also a fundamental difference between the schemes. The transformation in definition 8.2 does not use delay operations. So in the derivation, all the CSLD resolution steps following the first resolution step which introduces a constraint inconsistent with the added constraint are lost. The worst case for this transformation is when the first step of the transformation is unsatisfiable with the new constraint added. In this case all the information of the derivation is lost. On the other hand, Maher and Stuckey's method can address several derivations at a time. In our case, the resolution steps causing an inconsistency are delayed.

Whether incremental revision, backtracking or re-execution from scratch upon the addition of a constraint is a better strategy depends on the kind of applications considered. Our experiments with the application described in the previous section have shown a better behavior of the system under our incremental revision strategy, but it is clear that different conclusions can be drawn on different classes of applications. A key feature in our approach is the number of connected components in the constraint dependency graph which represents the dynamic interaction between constraints, and determines the impact of a revision.

9. CONCLUSION AND PERSPECTIVES

In the reactive constraint logic programming scheme we have proposed the possible interactions with the external world are defined through goal manipulation commands. The operational model of execution is based on a simple set of transformations of CSLD derivations and on a reactive search procedure. The capability of deleting constraints and atoms in a derivation has been used to define a new scheme for the addition of constraint to a query which, in contrast to other proposals, preserves the maximum information of a derivation by delaying derivation steps.

These operations have been implemented in our prototype reactive CLP(FD) system and have been evaluated on two different applications. The multi-frequency allocation problem illustrates the pertinence of the goal manipulation primitives to develop complex decision support systems. The on-line aircraft sequencing problem underlines the efficiency of the operational model. In the context of these applications, our incremental execution model has revealed better performances than re-execution from scratch, as well as its ability to localize revisions and to generate incrementally solutions which are close to the one preceding an interaction.

It is worth noting in this respect that the operations defined on CSLD derivations could also be useful for defining, in a general framework, non-backtracking search procedures for static CLP, such as solution repair, simulated annealing, tabu search, etc. The combination of these other search procedures with CLP is a major subject for extending the applicability of CLP to large scale combinatorial optimization problems.

The reactive CLP execution model is based on an incremental constraint solver with addition and deletion of constraints. We have shown that the presentation of this solver in the abstract framework of constraints as closure operators is faithful to constraint propagation algorithms and gives a simple proof of correctness of constraint relaxation.

Recent work on concurrent constraint programming (CC) such as on timed CC [25] or non-monotonic CC [10] [6] belongs to a similar line of research aiming at providing constraint programming with full reactive programming capabilities. One difference is that we have considered search problems, whereas in the CC approach for reactive systems, non-determinism is usually replaced by committed-choice indeterminism. On the other hand one simplification in our setting was the absence of dependencies due to constraint entailment checks [6].

Similar constraint retraction strategies have also been studied recently to handle over-constrained systems in [19] [24] [4]. In [4] a proposal is made to reduce the forward overhead of dependency maintenance, by choosing a different trade-off between efficiency and precision. Clearly the evaluation of these algorithms is delicate as the performances may vary a lot according to the characteristics of the problem at hand. Small benchmarks are thus not very conclusive in this domain, and more programming experiments on real problems will be needed to compare the choices of language constructs and the performances of their respective execution models.

Acknowledgements

This research was realized at the Central Research Lab. of Thomson-CSF in Orsay, with partial support from contract DRET 91 34 402. We are grateful to the many colleagues who participated to the project at some stage or another. The comments of the anonymous referees, which led to considerable improvement, are gratefully acknowledged.

REFERENCES

1. K.R. Apt. From chaotic iteration to constraint propagation. In *Proc. of 24th International Colloquium on Automata, Languages and Programming (ICALP '97), (invited lecture)*, Lecture Notes in Computer Science 1256, pages 36–55. Springer-Verlag, 1997.

2. C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *AAAI*, pages 221–226, Anaheim, USA, 1991.

3. P. Chatalic. Incremental techniques and prolog. Technical Report TR-LP-23, ECRC, ECRC GmbH, Arabellastr. 17, D-8000 Muenchen 81, Germany, 1987.

4. P. Codognet, D. Diaz, and F. Rossi. Constraint retraction in FD. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag LNCS 1116, 1996.

5. P. Codognet, F. Fages, and T. Sola. A metalevel compiler of clp(FD) and its combination with intelligent backtracking. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming Selected Research*, pages 437–456. MIT Press, 1993.

6. P. Codognet and F. Rossi. Nmcc programming: constraint enforcement and retraction in cc programming. In *ICLP'95*. MIT Press, 1995.

7. P. Codognet and T. Sola. Extending the wam for intelligent backtracking. In Koichi Furukawa, editor, *Proceedings of the eighth International Conference on Logic Programming*, pages 127–141, Paris, 1991. MIT Press.

8. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12. SIGPLAN Notices 12 (8), 1977.

9. B. De Backer and H. Beringer. Intelligent backtracking for clp languages: an application to clp(r). In *ILPS'91*. MIT Press, 1991.

10. F.S. De Boer, J.N. Kok, C. Palamidessi, and J.J. Rutten. Non-monotonic concurrent constraint programming. In *ILPS'93*. MIT Press, 1993.

11. R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *AAAI*, pages 37–42, 1988.

12. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–273, 1979.

13. F. Fages, J. Fowler, and T. Sola. Handling preferences in constraint logic programming with relational optimization. In *PLILP94*, Madrid, Spain, September 1994.

14. F. Fages, J. Fowler, and T. Sola. A reactive constraint logic programming scheme. In *ICLP'95*, Tokyo, June 1995. MIT Press.

15. D. Harel and A. Pnueli. On the development of reactive systems. In *NATO Advance Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, pages 477–498. Springer-Verlag NATO Series F, vol. 13, 1985.

16. J. Jaffar and J.L. Lassez. Constraint logic programming. In *POPL'87*, pages 111–119, Munich, January 1987. ACM.

17. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *JLP*, 19-20:503–581, May-July 1994.

18. J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. In M Bruynooghe and J Penjam, editors, *5th International Symposium, PLILP'93*, pages 60–74, Tallinn, Estonia, August 1993. LNCS 714, Springer-Verlag.

19. N. Jussien and P. Boizumault. Implementing constraint relaxation over finite domains using atms. In E. Freuder M. Jampel and M. Maher, editors, *Over-Constrained Systems*, pages 265–280. Springer-Verlag LNCS 1106, 1996.

20. J.L. Lassez. Parametric queries, linear constraints and variable elimination. In *DICS'90*, Capri, April 1990.

21. R. Lissajoux. *Meta(F) 3.58 users' manual*. Thomson-CSF, LCR, Orsay, 3.58 edition, 1994.

22. M. J. Maher and P. J. Stuckey. Expanding Query Power in Constraint Logic Programming Languages. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 20–36, Cleveland, Ohio, USA, 1989.

23. M.J. Maher. A logical semantics for a class of comitted choice languages. In J.L. Lassez, editor, *ICLP 87*, pages 858–876. MIT Press, may 1987.

24. F. Menezes and P. Barahona. Defeasible constraint solving. In E. Freuder M. Jampel and M. Maher, editors, *Over-Constrained Systems*, pages 151–170. Springer-Verlag LNCS 1106, 1996.

25. V.A. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *POPL'95*, January 1995.

26. V.A. Saraswat, M. Rinard, and P. Pranangaden. Semantic foundations of concurrent constraint programming. In *POPL'91: Proceedings 18th ACM Symposium on Principles of Programming Languages*, 1991.

27. Swedish Institute in Computer Science. *SICStus Prolog User's Manual*, 3.5 edition, October 1996.

28. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

29. P. Van Hentenryck. Incremental constraint satisfaction in logic programming. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 189–202, Jerusalem, 1990. The MIT Press.

30. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementations, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University, January 1993.

31. G. Verfaillie and T. Schiex. Solution reuse in dynamic CSPS. In *Proceedings AAAI*, Seattle, WA, August 1994.