A Reactive Constraint Logic Programming Scheme

F. Fages[1,2], J. Fowler[2] and T. Sola[2]

LIENS, CNRS[1]
Ecole Normale Supérieure
45, rue d'Ulm
75005 Paris, France
fages@dmi.ens.fr

LCR Thomson-CSF[2]
Domaine de Corbeville
91404 Orsay cedex, France.
{julian,sola}@thomson-lcr.fr

Abstract

In this paper we present a constraint logic programming scheme for reactive systems. A formal framework is developed to define the scheme's operational model and to prove its completeness. A prototype implementation of a simplified version of this model is described and then 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 [6]. Constraint logic programming (CLP) systems have been proved successful in complex system modeling and combinatorial optimization problems. A variety of applications have been developed over the last decade across various application domains, ranging from options trading and financial planning to job-shop scheduling, crew management, etc. [7].

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 who identified them as being particularly difficult systems. The reactive CLP systems we consider may not have severe 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 the domains below.

• 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 whilst

executing a schedule a problem such as machine failure may arise, thus requiring the adaptation of the current schedule.

The realisation of reactive and interactive systems within the CLP framework requires that the model of execution be extended with a mechanism to capture external events. One possible choice is to consider external events as query modification commands. Maher and Stuckey [11] defined an incremental execution model only for the *addition* of atoms and constraints to the query. Van Hentenryck [13] described methods of re-execution by oracle for the addition and deletion of constraints to a query. Neither of these methods, however, offers all the possibilities of incremental addition and deletion of both constraints and atoms to the query.

In this paper we study a reactive execution model for CLP, which allows all query manipulation commands. The reactive CLP scheme relies both on a system of transformation of SLD derivations and on the capability of the constraint solver to efficiently deal with addition and deletion of constraints. Such algorithms have already been proposed for dynamic constraint satisfaction problems [1] [4] [15]. In this paper we present a generic incremental constraint solver with addition and deletion of constraints in an abstract framework where constraints are identified to closure operators [12]. 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 notations on CLP languages. Section 3 presents the hypotheses under which our scheme is applicable to reactive systems. The formal model of execution is defined in section 4 where it is shown to be correct and complete w.r.t. the declarative semantics. The base of our current implementation is then described in section 5, and evaluated in section 6 on two applications: an interactive decision support system for frequency band allocation, and an on-line aircraft sequencing problem in a simulated environment.

2 Preliminaries and notations on CLP languages

A language of constraints 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 $b_1, b_2...$, 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 be seen also as finite multisets, where the *multiset union* of constraints c and d noted c, d denotes the conjunction of constraints, and *multiset difference* is noted $c \backslash d$ (*true* denotes the emptyset multiset of constraints).

A mathematical structure presented by a theory $\mathcal{D}$ is assumed to fix the interpretation of constraints. The $\mathcal{D}$-satisfiability of constraints is assumed to be decidable, i.e. one can decide 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*, that is the atoms are formed with variables only, constant and function symbols appear

exclusively in constraints.

CLP($\mathcal{D}$) programs are interpreted operationally 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) *[6] [10] 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 Hypotheses for a reactive CLP scheme

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.*

The principle hypothesis (hypothesis 1) is that 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 [11] is the *deletion* of constraints and atoms, not supported in their scheme.

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 information from the CSLD tree for the precedent 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 precedent 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 [2, 11, 13]. 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 [13], our execution model thus conserves only the information contained in the partially constructed CSLD tree of the precedent goal. Furthermore the transformation of the CSLD tree is based on a single derivation for the precedent goal, unlike [11]. 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 precedently. 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 hypothesis is about the structure of the problems that the scheme is best suited to solve.

Hypothesis 3 *Dense problems are considered such that few changes in the goal amount to revise a few number of steps in the* derivation *of a new solution.*

Therefore 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. We shall see that this capability can also be used to define new search procedures for static CLP problems.

4 Reactive execution model

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. A discussion of the associated search procedures will be given in the next section on implementation. The second part presents a generic incremental constraint solver with addition and deletion of constraints.

4.1 Transformation 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 solely on the transformation of that 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 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 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:*

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

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 formalises this idea, it marks the difference of our method with the methods of [11] and [13] for the constraint addition command.

Definition 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 by transformations over goals and derivations.

Definition 4 *Let* G *be a goal and* Δ *be a CSLD derivation from* G. *The goal manipulation commands are defined by the following transformations*[1]*:*

[1]Note that the addition of constraints may result in different transformations depending on the choice of a satisfiable subset of constraints. The choice of a satisfiable subset of constraints can be based on the dependency informations used by the incremental constraint solver presented in the next section. In our context a notion of maximally satisfiable subset should take into account the dependencies between resolution steps in the derivation.

$< c, d\|\alpha;\ \Delta >$	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)	$< c\|\alpha, \beta;\ \Delta \oplus \alpha >$

One can easily check that the transformations define correct CSLD derivations for the transformed goals. The execution model consists of developing a CSLD tree for the modified goal containing the transformed derivation. The completeness of the scheme is a corollary of the following:

Proposition 4.1 (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 1 *The transformation for the addition of a constraint can be illustrated by the following typical disjunctive scheduling CLP program over natural integers:*

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

The following goal has a 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. The command *add_constraint*$(2 \geq m)$ can compute the following transformed derivation by delaying the first resolution step.

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

The search continues from that derivation giving a success in one transition.

4.2 Incremental constraint solving with addition and deletion of constraints

The practicality of the previous CSLD tree transformation system depends on the ability of the constraint solver to efficiently deal with addition and deletion of constraints. In this section, we present in an abstract way a general scheme for dynamic constraint solving, and illustrate the kind of algorithm obtained with an example over finite domains.

In this section we shall identify a constraint solver with a closure operator over a structure of domain variables. We introduce a labelled transition system where each transition is labelled by an atomic constraint. The solving of a system of atomic constraints is then defined by the derivations obtained in this system.

An *environment* $E : \mathcal{V} \rightarrow 2^{\mathcal{D}}$ associates a domain of possible values to each variable. Environments form a lattice structure $(\mathcal{E}, \sqsubseteq)$ for the *information ordering* defined by $E \sqsubseteq E'$ iff $\forall x \in \mathcal{V}\ E(x) \supseteq E'(x)$. Note the duality between the information ordering and the domain ordering, the union of information corresponds to the intersection of domains.

Now let $\mathcal{C}$ be the set of atomic constraints $\{b_i\}_{1 \leq i \leq n}$. The semantics of each atomic constraint b is supposed to be defined as a closure operator over $\mathcal{E}$ [12], noted $\overline{b}$, i.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) $\overline{b}(\overline{b}(E)) = \overline{b}(E)$.

As is well known the union of closure operators is not a closure operator, however one can define for a constraint c (i.e. a system of atomic constraints) the closure operator

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

which models the conjunction of atomic constraints in c. Now the method of approximating $\overline{c}$ by iterating the atomic constraint operators $\overline{b}$ for $b \in c$ is faithful to constraint propagation algorithms for solving systems of constraints (note however that termination is not assumed at this stage).

Proposition 4.2 (Correctness of constraint propagation) *Let c be a system of atomic constraints $b_1, ..., b_n$. Let E be an environment. Then $\overline{c}(E)$ is the limit of any fair iteration of closure operators $\overline{b}_1, ..., \overline{b}_n$ from E.*

Incremental constraint solving can thus be modeled as a transition system defined by atomic constraints. A labelled transition system $< \Gamma, \longrightarrow >$ is defined where Γ is the set of configurations and $\longrightarrow \subseteq \Gamma \times \mathcal{C} \times \Gamma$ is the transition relation labelled by atomic constraints. For dealing moreover with deletion of constraints, informations about dependencies need be maintained. Consequently a configuration will be a triple

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

composed of an environment E, a *producer* function P which associates with an atomic constraint and a variable the set of domain values removed by the constraint from the domain of the variable, and a *consumer* function C which associates with a variable the set of basic constraints which use the domain of this variable to reduce other domains[2].

The transition system is defined by the following constraint propagation rule (CP):

$$CP\ \frac{b_i \in \mathcal{C} \qquad E' = \overline{b_i}(E) \qquad E' \neq E}{< E, P, C > \xrightarrow{b_i} < E', P', C' >}$$

where $P'(b_i, x) = P(b_i, x) \cup (E(x) \setminus E'(x))$, and $C'(x) = C(x) \cup \{b_i\}$ if $x \in \mathcal{V}(b_i)$ and $E'(y) \neq E(y)$ for some variable $y \in \mathcal{V}(b_i) \setminus \{x\}$, $C'(x) = C(x)$ otherwise.

[2]Note that this definition does not preclude an explicit representation of domains in the implementation.

As a corollary of proposition 4.2 we get that for a system of constraints c, a terminating sequence of transitions labelled by atomic constraints in c from an initial environment E ends with final environment $\overline{c}(E)$[3].

The labelled transition system defines a generic dependency graph structure. This structure can be used to build a graph for intelligent backtracking [3]. In that case, only the constraints responsible for the insatisfiability need to be determined, so only the *producer* function is required. The structure can be useful also for a debugging environment. For our purpose we shall use the *constraint dependency graph* of a configuration $< E, P, C >$ defined as the graph of atomic constraints such that there is an arc from b to b' if and only if there exists a variable $x \in \mathcal{V}(b)$ such that $P(b, x) \neq \emptyset$ and $b' \in C(x)$.

In the incremental constraint solver the operation of *addition of a constraint*, noted `add(c)`, simply adds c to the system of constraints and applies the transitions up to a fixed point. The operation of *deletion of an atomic constraint*, noted `del(b)`, deletes b from the system of constraints, and applies the transitions up to a fixpoint from the environment obtained by the operation of *constraint relaxation* relax(b) defined formally as follows:

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

The environment obtained by relaxation may not be attainable by constraint propagation. However an immediate consequence of the fact that a constraint is a closure operator is that:

Proposition 4.3 *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 is comprised between the initial environment and its fixed point for the constraint system without the deleted atomic constraint.

Lemma 4.4 *Let $< E_0,\ P_0,\ C_0 > \xrightarrow{b_1} \ \ldots \ \xrightarrow{b_n} < E_n,\ P_n,\ C_n >$ be a finite transition sequence with constraint system c. Let $< c,\ E_n,\ P_n,\ C_n\ >$ relax(b) $< c \setminus b,\ E,\ P,\ C >$ be the configuration obtained after the deletion of atomic constraint b. Then $E_0 \sqsubseteq E \sqsubseteq \overline{c \setminus b}(E_0)$.*

Proposition 4.3 and lemma 4.4 together show the soundness of the incremental constraint solver with addition and deletion of constraints[4].

Theorem 4.5 *Let E' be an environment obtained by the incremental constraint solver from an initial environment E, by applying a sequence of addition and deletions of constraints resulting in a constraint system c. Then $E' = \overline{c}(E)$.*

[3] For the sake of brevity the failure environments which assign an empty domain to a variable are not distinguished. Unsatisfiable systems of constraints could be detected earlier in that case by defining the transition to a *fail* configuration.

[4] Note that unless constraint propagation may be interrupted by constraint deletion commands, in the use of theorem 4.5 E' is a fixed point of $\overline{c}$. In this case not all atomic constraints in $c' = c \setminus b$ need be propagated after the relaxation of b. The information contained in the producers and consumers of the configuration can be used to determine a subset of constraints in c' which are necessary to propagate.

Example 2 *We give an example of a constraint system on interval arithmetic. We consider the transition rule associated to the constraints of the form* $x \geq y + cst$, *where* $x, y \in \mathcal{V}$, *and cst is an integer, that models interval propagation.*

Consider the system of three constraints : $y \geq z + 1$, $y \geq x + 1$ *and* $t \geq z + 1$ *noted* b_1, b_2 *and* b_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 consider a transition sequence $< E_0,\ P_0,\ C_0 > \xrightarrow[b_1,b_2,b_3]{*} < E_3,\ P_3,\ C_3 >$ *where* $E_3(x) = [1, 9]$; $E_3(y) = [2, 10]$; $E_3(z) = [1, 8]$; $E_3(t) = [2, 9]$

$P_3(b_2,\ x) = \{10\}$; $P_3(b_1,\ z) = \{10\}$; $P_3(b_3,\ z) = \{9\}$; $P_3(b_3,\ t) = \{1\}$

$C_3(y) = \{b_1, b_2\}$; $C_3(z) = \{b_3\}$; $C_3(t) = \{b_3\}$.

Now if the constraint b_1 *is deleted we obtain*

$< c,\ E_3,\ P_3,\ C_3 >\ relax(b_1) < c \setminus b_1,\ E,\ P,\ C >$

where $E(x) = [1, 9]$; $E(y) = [2, 10]$; $E(z) = [1, 10]$; $E(t) = [1, 9]$, $P(b_2,\ x) = \{10\}$ *and* $C(y) = \{b_2\}$.

The constraint solver then terminates in a single transition

$< E,\ P,\ C > \xrightarrow{b_3} < E_4,\ P_4,\ C_4 >$ *with*

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

$P_4(b_2,\ x) = \{10\}$; $P_4(b_3,\ z) = \{9, 10\}$; $P_4(b_3,\ t) = \{1\}$

$C_4(y) = \{b_2\}$; $C_4(z) = \{b_3\}$; $C_4(t) = \{b_3\}$.

5 Implementation

5.1 Simplifying hypothesis on non-determinism

In our current implementation with Meta(F) [3] the constraint solvers are written in C and interfaced with Sicstus Prolog through the standard efficient interface. The constraint dependencies are fully managed by the constraint solvers whilst the dependencies among atoms are eliminated by a last simplifying hypothesis.

Hypothesis 4 *Deletable atoms are supposed to be defined by clauses whose body contains no atom only constraints.*

Deletable atoms are thus supposed to be defined by a set of program clauses of the form $\{A \leftarrow c_1 \mid \Box, ..., A \leftarrow c_n \mid \Box\}$. This form is typical of disjunctive constraints that represent the combinatorial aspects of the problem. This hypothesis simplifies the deletion of atoms, because the resolution of an atom does not introduce supplementary atoms.

5.2 Reactive search procedure

We consider the possible strategies a reactive constraint logic programming scheme might adopt in its search of a CSLD tree. We recall that a derivation of the CSLD tree is transformed by a goal manipulation to give a new derivation, which is the point of departure for the development of a CSLD tree for the new goal. A reactive search procedure explores an entire CSLD tree from an initial derivation, which is the result of the goal manipulation. The incremental constraint solver with addition and deletion of constraints allows us to implement such reactive search procedures with a simple meta-interpreter.

5.2.1 Iterative search vs backtracking for static CLP problems

Before considering the reactive search procedure we illustrate the flexibility acquired by the presence of an explicit operation for removing a constraint from the store (operation `del`). The iterative search procedure traverses a CSLD tree in a depth first, left to right order without backtracking. Figure 1 presents a Prolog meta-interpreter for iterative search.

```
search([]):-success.
search([(A<-c,A<-d)|Goal]):-
       (add(c)->search(Goal),del(c);true),
       (add(d)->search(Goal),del(d);true).
```

Figure 1: Prolog Meta-Interpreter for iterative search

The meta-interpreter is called with the predicate `search`. The argument is a goal given as a list of deletable atoms written with the clauses defining them. Here we only consider deletable atoms with two rules `A<-c, A<-d`. If the derivation is successful the predicate `success` is called, otherwise the search continues through `del` and `add` operations. Note that iterative search need not be limited to the simple simulation of backtracking search procedures for static CLP programs. It is possible to implement in this way heuristic driven search procedures which may be non-exhaustive, based on solution repair, simulated annealing, etc.

5.2.2 Reactive search for dynamic CLP problems

The reactive search procedure takes into account interactions and combines the iterative search procedure (figure 1) with the derivation transformations presented in section 4. The reactive search meta-interpreter given in figure 2 is called by the predicate `react` which takes two arguments. The first argument, `G`, is the goal given as a list of atoms written with the clauses defining them. The second argument, `D`, is the current derivation given in the same form. The predicate `goal_modification`

```
react(G,D):-
       (goal_modification(M)->
       transformation(M,G,D,G1,D1),
       search(G1,D1),
       back_search(G1,D1);
       search(G,D)).

search([],Der):-success(Der).
search([(A<-C,A<-D)|G],Der):-
       (add(C)->react(G,[(A<-C,A<-D)|Der]),del(C);true),
       (add(D)->react(G,[(A<-D,A<-C)|Der]),del(D);true).

back_search(G,[]):-failed(G).
back_search(G,[(A<-C,A<-D)|Der]):-
       del(C),
       (add(D)->search(G,[(A<-D,A<-C)|Der]),del(D);true),
       back_search([(A<-C,A<-D)|G],Der).
```

Figure 2: Prolog Meta-interpreter for reactive search

takes into account interactions. The predicate `transformation` transforms the goal

and the derivation as described in section 4.

After a goal modification `react` first searches for a successful derivation from the current derivation using the predicate `search` which is similar to the one defined in the previous subsection. If a successful derivation is found then the predicate `success` is called to signal the success and wait for interactions.

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 `back_search` explores the other portions of the CSLD tree by remounting the derivation.

Note that at any point the search process may be interrupted and the goal modified, starting a new search from the transformed derivation. For the sake of simplicity this meta-interpreter guards continuations which may be abandonned.

6 Evaluation

6.1 Multi-Frequency Allocation

We have used the model in an application 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 3. 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. In addition to the multi-criteria optimization techniques described in [5] 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 operator may interact with the problem in any one of the following ways. The interactions are the composition of several goal manipulations.

• Increase the number of frequencies `NFj` of a selected network `j` (the interaction "better freq" of figure 3). The basic idea is to add the constraint, `add_constraint( NFj > cj)`, to the goal stating that frequencies for the new allocation on the selected network must be superior to the number 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 introduction of this constraint from 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

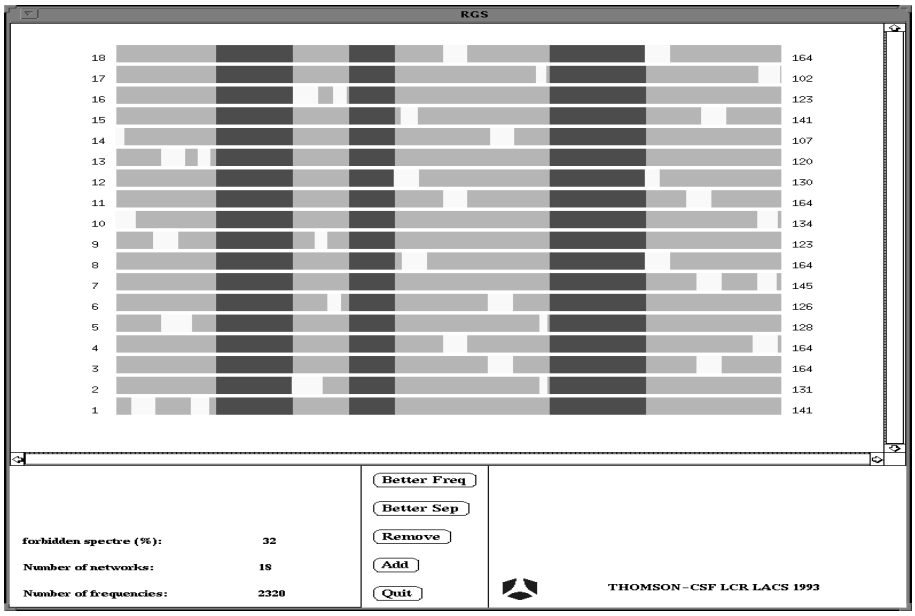

Figure 3: Multi-frequency allocation screen dump

the interaction will be superior 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 commences its exploration for a new successful derivation with the hard constraint of not decreasing the quality of the allocation to the other networks. The operator sees only the modification of frequency allocation of certain networks. In fact these networks correspond to the constraints removed to "cure" the insatisfiability of the derivation.

• Increase the separation of the two bands of a selected network `j` the interaction "better sep". This is similar to the interaction "better freq".

• Modify the forbidden frequency bands. 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`).

In a typical scenario an initial allocation is 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 has for effect to perturb a solution in such a way that the solution found after an interaction is "close" to the solution before the interaction. This process continues until a good allocation is found, this allocation is then updated when the forbidden frequencies change and the new allocation is refined if necessary.

6.2 Aircraft sequencing

The terminal zone aircraft sequencing problem (ASP) represents today an important bottleneck of air traffic flow management. Its statement might be summarized

as follows: given a set of aircraft entering in the terminal area (e.g. 30 mn from the airport) determine an optimal sequence, according to aircraft capacities, procedural safety constraints, terminal configurations, 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 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 STA_x for each flight x. These variables are constrained by the expected time of arrival ETA_x. The duration of the task models the safety distance between flights that is a function of the particular aircrafts in the sequence. The possible permutations of the flights are constrained by the position of the flights in the air corridors. The details of the modelization can be found in [8]. Here we focus on the possible interactions supported by the system 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 requires also the addition of a disjunctive constraint atom (`add_atom(disj(STAx,STAy,Dx,Dy))`)[5] to the goal for each flight in the terminal zone which can be permuted with the added flight.

• The removal of a flight is the complement of the addition of a flight. It 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 the addition of constraints to the goal, 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 in this framework is modeled as a particular case of interaction with an agent which repeatedly constraints the final scheduled time of arrival of the sequence.

Table 2 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). 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 resolution* respectively. Times are also presented for finding an optimal sequence after each evolution of

[5]The predicate `disj` could be defined as in example 1 but for efficiency reasons it is actually defined by constructive disjunction [9] [14].

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	**16.0**	**86.2**
3	149	20	3839	10	**7.45**	**384**
4	160	10	10050	88	**16.0**	**114**
5	140	1510	3949	4380	**0.09**	**0.90**
6	129	20	3907	60	**6.45**	**65.1**
7	99	20	9425	528	**4.95**	**17.9**
8	99	10	8050	60	**12.0**	**134**
9	99	10	5179	10	**14.0**	**518**

Table 2: Computation times in milliseconds for aircraft sequencing

the problem w.r.t. the static approach (S_{opt}) and the reactive one (R_{opt}). 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.

7 Conclusion

In the constraint logic programming scheme we have proposed for reactive systems the possible interactions are defined through goal manipulation commands. The operational model of execution is based on transformations of CSLD derivations and on the concept of reactive search. Our scheme supports all goal manipulations, whilst [11] supports only addition of constraints and atoms. The capability of deleting constraints and atoms in a derivation is used also to define a new scheme for the addition of constraint to a query which in contrast to [13], preserves the maximum information of a derivation by delaying derivation steps.

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.

The reactive CLP scheme has 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 we propose.

Note finally that the operations on CSLD derivations we have defined may also serve as a general framework to define non-backtracking search procedures for static CLP, such as solution repair, simulated annealing, tabu search, etc. This will be the matter for future experiments.

Acknowledgements: This research was supported in part by the French Ministry of Defence under contract DRET 91 34 402.

References

[1] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *AAAI*, 1991.

[2] Philippe Chatalic. Incremental techniques and prolog. Technical Report TR-LP-23, ECRC GmbH, Arabellastr. 17, D-8000 Muenchen 81, Germany, 1987.

[3] 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.

[4] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *AAAI*, pages 37–42, 1988.

[5] François Fages, Julian Fowler, and Thierry Sola. Handling preferences in constraint logic programming with relational optimization. In *PLILP94*, Madrid, Spain, September 1994.

[6] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL'87*, pages 111–119, Munich, January 1987. ACM.

[7] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *JLP*, 19-20:503–581, May-July 1994.

[8] J. Jourdan. Modelisation of terminal zone aircraft sequencing in clp. Technical Report LACS-92-6, LCR Thomson-CSF, Oct 1992.

[9] J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. In *PLILP'93*, pages 60–74, Tallinn, Estonia, August 1993.

[10] 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.

[11] M.J. Maher and P.J. Stuckey. Expanding query power in constraint logic programming languages. In E. Lusk and R. Overbeek, editors, *NACLP89*, pages 20–36. MITP, oct 1989.

[12] Vijay A. Saraswat. Concurrent constraint programming. In *POPL'91: Proceedings 18th ACM Symposium on Principles of Programming Languages*, 1991.

[13] P. Van Hentenryck. Incremental constraint satisfaction in logic programming. In David H. D. Warren and Piter Szeredi, editors, *ICLP'90*, pages 189–202, Jerusalem, 1990. The MIT Press.

[14] 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.

[15] G. Verfaillie and T. Schiex. Solution reuse in dynamic CSPS. In *Proceedings AAAI*, 1994.