Design,Implementations,and Evaluation of the Constraint Language cc(FD)

Pascal Van Hentenryck[1]
Vijay Saraswat[2]
Yves Deville[3]

Technical Report No. CS-93-02

January 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]Xerox Palo alto Research Center 3333 Coyote Hill Road, Plao Alto, CA 94304
[3]Universite Catholique de Louvain, Pl. Ste Barbe,2 B-1348 Louvain, Belgium

Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`

Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

Vijay Saraswat
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304
Email: saraswat@parc.xerox.com

Yves Deville
Université Catholique de Louvain, Pl. Ste Barbe, 2 B-1348 Louvain-La-Neuve, Belgium
Email: yde@info.ucl.ac.be

Abstract

This paper describes the design, implementation, and applications of the constraint logic language `cc(FD)`. `cc(FD)` is a declarative nondeterministic constraint logic language over finite domains based on the `cc` framework [27], an extension of the CLP scheme [16]. Its constraint solver includes (non-linear) arithmetic constraints over natural numbers which are approximated using domain and interval consistency. The main novelty of `cc(FD)` is the inclusion of a number of general-purpose combinators, in particular cardinality, constructive disjunction, and blocking implication, in conjunction with new constraint operations such as constraint entailment and generalization. These combinators significantly improve the operational expressiveness, extensibility, and flexibility of CLP languages and allows issues such as the definition of non-primitive constraints and disjunctions to be tackled at the language level. The implementation of `cc(FD)` (about 40,000 lines of C) includes a WAM-based engine [36], optimal arc-consistency algorithms based on AC-5 [34], and incremental implementation of the combinators. Results on numerous problems, including scheduling, resource allocation, sequencing, packing, and hamiltonian paths are reported and indicate that `cc(FD)` comes close to procedural languages on a number of combinatorial problems. In addition, a small `cc(FD)` program was able to find the optimal solution and prove optimality to a famous 10/10 disjunctive scheduling problem [23], which was left open for more than 20 years and finally solved in 1988.

1 Introduction

Constraint Logic Programming (CLP) is a new class of declarative programming languages combining nondeterminism and constraint solving. The fundamental idea behind these languages, to use constraint solving instead of unification as the kernel operation of the language, was elegantly captured in the CLP scheme [16]. The CLP scheme can be instantiated to produce a specific language by defining a constraint system (i.e. defining a set of primitive constraints and providing a constraint solver for the constraints). For instance, `CHIP` contains constraint systems over finite domains [30], Booleans [3] and rational numbers [14, 35], Prolog III [9] is endowed with constraint systems over Booleans, rational numbers, and lists, while CLP($\Re$) [17] solves constraints over real numbers. The CLP scheme was further generalized into the cc framework of concurrent constraint programming [27, 28, 29] to accommodate additional constraint operations (e.g. constraint entailment [21]) and new ways of combining them (e.g. implication or blocking ask [27] and cardinality [32]).

CLP languages[1] support, in a declarative way, the solving of combinatorial search problems using the global search paradigm. The global search paradigm amounts to dividing recursively a problem into subproblems until the subproblems are simple enough to be solved in a straightforward way. The paradigm includes, as special cases, implicit enumeration, branch and bound, and constraint satisfaction. It is best contrasted with the local search paradigm, which proceeds by modifying an initial configuration locally until a solution is obtained. These approaches are orthogonal and complementary. The global search paradigm has been used successfully to solve a large variety of combinatorial search problems with reasonable efficiency (e.g. scheduling [5], graph coloring [18], Hamiltonian circuits [8], microcode labeling [12]) and provides, at the same time, the basis for exact methods as well as approximate solutions (giving rise to the so-called "anytime algorithms" [11]).

CLP languages over finite domains (e.g. CHIP [13, 30]) have been applied to numerous discrete combinatorial problems, including graph coloring, cutting stock, microcode labeling, warehouse location, and car-sequencing. For many problems, they allow a short development time and an efficiency which compares well with procedural languages implementing the same approach. For other problems however, the CLP scheme appears to lack flexibility and operational expressiveness since it only offers constraint solving over a fixed set of predefined constraints. As a consequence, many problems lose their natural formulation and need to be recasted in terms of more basic variables and constraints, inducing a significant loss in efficiency.

The research described in this paper is an attempt to overcome some of the limitations of CLP languages while preserving their benefits: short development time and referential transparency. It describes `cc(FD)`, an instance of the cc framework over finite domains.

The main novelty in the design of `cc(FD)` is the inclusion of a number of general purpose combinators, i.e. cardinality, constructive disjunction, and blocking implication. The combinators are *general-purpose* in the sense that they apply to any constraint system and are not tailored to the constraint system of `cc(FD)` and *declarative* since they preserve referential transparency. In conjunction with new constraint operations such as constraint entailment and generalization, the new combinators significantly enhance the operational expressiveness and efficiency of CLP languages and enable us to address issues such as the definition of non-primitive constraints and the handling of disjunctions at the language level. As a consequence, the combinators, together with a small and natural set of constraints over integers, preclude the need for many ad-hoc extensions which were introduced for efficiency reasons but were difficult to justify from a theoretical standpoint. `cc(FD)` preserves or improves the efficiency of problems previously solved by CLP languages over finite domains but also allows the solving of problems that were previously out of scope for CLP languages, e.g. resource allocation and disjunctive scheduling problems. In particular, we were able, using `cc(FD)`, to find the optimal solution, and prove its optimality, to a famous 10/10 scheduling problem [23], which was left open for more than 20 years and finally solved in 1988 [5].

The key novelties in the implementation of `cc(FD)` (about 40,000 lines of C) are the inclusion of optimal consistency algorithms based on AC-5 [34], dynamic specializations of data structures

[1] In the following, we use the term *CLP languages* generically to denote both CLP and cc languages.

```
sizeMaster(112).
sizeSquares([50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2]).
```

Figure 1: The Data for the Perfect Square Problem

and constraints, and incremental algorithms for the combinators.
The contributions of this paper are as follows:

1. it presents cc(FD), a simple, uniform, and clean declarative nondeterministic constraint logic language over finite domains;

2. it demonstrates, by means of simple examples, programming idioms to design non-primitive constraints and pruning techniques without resorting to ad-hoc extensions;

3. it discusses how cc(FD) can be implemented to obtain a efficient performance;

4. it gives experimental results which indicate the viability of this approach, even for very complex problems such as the famous 10/10 scheduling problem.

The rest of this paper is organized as follows: the first section presents a motivating example, the perfect square problem, to acquaint the reader with the programming style in cc(FD). The next section discusses the design of cc(FD), including the constraint solver, the combinators, and some higher-order predicates. Section 4 discusses the implementation while the last section reports a large number of experimental results.

2 A Motivating Example

To illustrate several features of cc(FD), we present a program to solve the so-called perfect square problem. The purpose of the program is to build a square, called the *master square*, out of a number of given squares. All the squares must be used and they all have different sizes. The squares are not allowed to overlap and no empty space is permitted in the master square. The sizes of the squares (i.e. the size of their side) and the size of the master squares are given and are depicted in Figure 1. This problem is very combinatorial and there is no hope to solve it using simple backtracking approaches. An interesting fact is that 21 is the smallest number of squares, all of different sizes, which can be packed to produce a master square.

Most programs in cc(FD) follow the following schema

```
solveProblem(...) :-
   generateVariables(...),
   stateConstraints(...),
   stateSurrogateConstraints(...),
   makeChoices(...).
```

The first goal in the body simply creates the problem variables and specifies their ranges. The second goal states the problem constraints. Since, in general, the constraint solver only approximates the constraints, the last goal makes nondeterministic choices to obtain a solution. The third goal states surrogate constraints, i.e. constraints expressing properties of the solutions. These constraints are redundant from a semantic standpoint but are fundamental from an operational standpoint since they may dramatically reduce the search space. This is a traditional technique in operations research. For the perfect square problem, the top-level predicate is as follows:

```
packSquares(Xs,Ys) :-
   generateSquares(Xs,Ys,Sizes,Size),
   stateNoOverlap(Xs,Ys,Sizes),
   stateCapacity(Xs,Sizes,Size), stateCapacity(Ys,Sizes,Size),
   labeling(Xs), labeling(Ys).
```

The first goal generates the lists of variables `Xs` and `Ys` of x and y coordinates of all squares, a list `Sizes` with the given sizes of all squares, and the given size `Size` of the master square. The goal `stateNoOverlap` states the no-overlapping constraints while the goals `stateCapacity` state surrogate constraints exploiting the fact that there is no empty space. The last two goals are nondeterministic goals to generate values for the coordinates. We now study these procedures in more detail.

Each square i is associated with two variables X_i and Y_i representing the coordinates of the bottom-left corner of the square. Each of these variables ranges between 0 and $S - S_i$ where S is the size of the master square and S_i is the size of square i. The following procedure describes the creation of the two lists of variables as well as the list of the sizes.

```
generateSquares(Xs,Ys,Sizes,Size) :-
   sizeMaster(Size), sizeSquares(Sizes),
   generateCoordinates(Xs,Ys,Sizes,Size).

generateCoordinates([],[],[],_).
generateCoordinates([X|Xs],[Y|Ys],[S|Ss],Size) :-
   MaxCoord := Size - S, X ∈' 0..MaxCoord, Y ∈' 0..MaxCoord,
   generateCoordinates(Xs,Ys,Ss,Size).
```

The no-overlap constraint between two squares `(X1,Y1,S1)` and `(X2,Y2,S2)` where `(X1,Y1)` and `(X2,Y2)` are the positions of the squares and `S1` and `S2` are their respective sizes can be expressed using constructive disjunction, one of the combinators of `cc(FD)`:

```
nooverlap(X1,Y1,S1,X2,Y2,S2) :-
   X1 + S1 ≤'' X2 ∨' X2 + S2 ≤'' X1 ∨' Y1 + S1 ≤'' Y2 ∨' Y2 + S2 ≤'' Y1.
```

The disjunction simply expresses that the first square must be on the left, on the right, below, or above the second square. Operationally, `cc(FD)` removes all values not satisfied by any of the disjuncts (in conjunction with the accumulated constraints) from the domain of the variables.

There is no need to state the no-empty space constraint thanks to the domain of the coordinates, the no-overlap constraint and the hypothesis that the surface of the master square is equal to the sum of the areas of the squares.

A traditional technique to improve efficiency in combinatorial search problem amounts to exploiting properties of all solutions by adding redundant or surrogate constraints. In the perfect square problem, the sizes of all squares containing a point with a given x-coordinate (resp. y-coordinate) must be equal to S, the size of the master square, since no empty space is allowed. These surrogate capacity constraints can be stated using cardinality and linear equations. For a given position P, the idea is to associate with each square i a boolean variable Bi (i.e. a 0-1 variable) that is true iff square i contains a point with x-coordinate (resp. y-coordinate) P. The boolean variable is obtained using the cardinality operator of cc(FD), i.e.

```
#(Bi,[Xi ≤" P #& P ≤" Xi + Si - 1 ],Bi).
```

A cardinality formula $\#(l, [c_1, \ldots, c_n], u)$ states that the number of formula which are true in $\{c_1, \ldots, c_n\}$ is no less than l and no more than u. Operationally, the cardinality uses constraint entailment to find out if there is a way to satisfy the constraint and constraint solving when there is a unique way to satisfy the formula. The surrogate constraint for position P and the x coordinate can now be stated as a simple linear equation:

```
B1 * S1 + ... + Bn * Sn =" Size.
```

The program to generate a surrogate constraint is as follows:

```
capacity(Position,Coordinates,Sizes,Size) :-
   accumulate(Coordinates,Sizes,Position,Summation),
   Summation =" Size.

accumulate([],[],_,0).
accumulate([C|Cs],[S|Ss],P,B*S + Summation) :-
   B ∈' 0..1,
   #(B,[ C ≤" P #& P ≤" C + S - 1],B),
   accumulate(Cs,Ss,P,Summation).
```

The generation of places for the squares requires to give values to the x and the y coordinates of all squares. We use the idea of [1] for the labeling of a coordinate, exploiting the fact that no empty space is allowed. At each step, the program identifies the smallest possible coordinate and select a square to be placed at this position. On backtracking, another square is selected for the same position. The labeling is as follows:

```
labeling([]).
labeling([Coord|Coords]) :-
   minlist([Coord|Coords],Min),
   selectSquare([Coord|Coords],Min,Rest),
   labeling(Rest).

selectSquare([Coord|Coords],Min,Coords) :-
   Coord =" Min.
selectSquare([Coord|Coords],Min,[Coord|Rest]) :-
   Coord >" Min,
   selectSquare(Coords,Min,Rest).
```

The first goal in the labeling finds the smallest position for the remaining squares while the second goal chooses a square to assign to the position. Since no empty space is allowed, such a square must exist.

This concludes our motivating example. As is easily shown, the program is rather small and about one page long. It packs 21 or 24 squares in a master square in about 30 seconds on a Sun Sparc Station, illustrating the expressiveness and efficiency of the language.

3 The Design of `cc(FD)`

We now turn to the design of `cc(FD)`. `cc(FD)` is a small and uniform language, based on a small constraint system (from a conceptual standpoint) and a number of general-purpose combinators. The key contribution is of course the inclusion of the new combinators and their associated constraint operations. The novelty in the constraint solver is its simplicity and the explicit distinction between domain and interval reasoning, two techniques that were previously hidden in the implementation. This section reviews the various aspects of the design of `cc(FD)`.

3.1 The Constraint System

3.1.1 Syntax and Semantics

In this section, we describe the functionality of the constraint system of `cc(FD)`. We focus on finite domains and omit the traditional constraints on first-order terms.

Primitive constraints in `cc(FD)` are built using variables, natural numbers, the traditional integer operators $+, -, *, div, mod$ and the relations

$$>', \geq', =', \neq', \leq', <', \quad \text{and} \quad >'', \geq'', ='', \neq'', \leq'', <'' .$$

div and *mod* represent the integer division and remainder. The arithmetic relations are duplicated to make explicit the two forms of reasoning used in the constraint solver: domain consistency (single quote) and interval consistency (double quote). The former are called *domain* constraints and the latter *interval* constraints. Variables appearing in constraints are assumed to take values from a finite set of natural numbers, e.g. the set of natural numbers that can fit in a memory word. For convenience, `cc(FD)` also provides the range constraints

$$x \in' [a_1, \ldots, a_n], \;\; x \in' l..u, \;\; x \notin' [a_1, \ldots, a_n], \;\; x \notin' l..u$$

and

$$x \in'' [a_1, \ldots, a_n], \;\; x \in'' l..u, \;\; x \notin'' [a_1, \ldots, a_n], \;\; x \notin'' l..u$$

although they can easily be obtained from the previous constraints in conjunction with the combinators. Note that the negation of a constraint is also a constraint. In the following, we use the term *constraint store* to denote a conjunction of constraints and use σ possibly subscripted to denote constraint stores. It should be clear that a computation state in `cc(FD)` is a pair $\langle B, \sigma \rangle$ where B is a conjunction of goals that remain to be solved and σ is a constraint store representing all constraints accumulated up to that point. The above constraints are also called primitive constraints. We will see that the combinators allow us to define new (non-primitive) constraints.

3.1.2 Constraint Operations

As mentioned previously, the combinators of `cc(FD)` are general-purpose and not tailored to the above constraint system.[2] They use three operations on a constraint system $\mathcal{C}$:

1. **constraint solving**: deciding the consistency of a constraint store σ, i.e. $\mathcal{C} \models (\exists)\sigma$;
2. **constraint entailment**: deciding whether a constraint c is entailed by a constraint store σ, i.e. $\mathcal{C} \models (\forall)(\sigma \Rightarrow c)$;
3. **constraint generalization**: finding a generalization σ of a set of constraint stores $\{\sigma_1, \ldots, \sigma_n\}$, such that
$$C \models (\forall)(\sigma_i \Rightarrow \sigma) \quad (1 \leq i \leq n) \qquad (1)$$

For constraint generalization, we would like in general the strongest constraint σ satisfying property (1). This is given, for instance, by the *lub* operation when the constraint system is a complete lattice with respect to the implication order on constraints. Many constraint systems do not enjoy the existence of a *lub* but any constraint store satisfying property (1) is sufficient.

3.1.3 Constraint Processing in `cc(FD)`

Constraint solving and constraint entailment are decidable problems for `cc(FD)` (since only a finite set of integers is considered) but they are NP-complete problems. For this reason, `cc(FD)` approximates them by using domain and interval reasoning. The main idea behind domain reasoning is to use constraints to remove values from the domains, to use the domains to decide constraint entailment, and to generate membership constraints during generalization.[3] The main idea behind interval reasoning is to use constraints to reduce the lower and upper bounds on the domains, to use the bounds to detect entailment, and to generate new bounds during generalization. The purpose of the next two sections is to describe the solver of `cc(FD)` in a precise way. For simplicity, we assume that all constraints are implicitly defined on a set of variables $\{x_1, \ldots, x_n\}$.

3.1.4 Domain Reasoning

Domain reasoning is applied on the domain constraints $>', \geq', =', \neq', \leq', <', \in', \notin'$. Instead of checking consistency of these constraints, `cc(FD)` checks domain satisfiability, i.e. it enforces domain consistency and checks if none of the domains is empty. The key idea is to associate with each variable its possible set of values. We now define the three operations for domain reasoning: domain consistency, domain entailment and domain generalization.

Definition 1 A constraint c is *domain-consistent* wrt $D_1, \ldots, D_n$ if, for each variable x_i and value $v_i \in D_i$, there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $c(v_1, \ldots, v_n)$ holds. A constraint store σ is *domain-consistent* wrt $D_1, \ldots, D_n$ if any constraint c in σ is domain-consistent wrt $D_1, \ldots, D_n$.

In `cc(FD)`, domain consistency is achieved in an incremental way by reducing the domains of the variables at each computation step.

[2] Our current design and implementation efforts are devoted to build `cc(Q)` and `cc(B)`, two instances of the same framework for rational linear arithmetics and Boolean algebra.

[3] The use of domain consistency in programming language was suggested first by Mackworth [20].

Definition 2 The *reduced domains* of a constraint store σ are the largest domains $D_1, \ldots, D_n$ such that σ is domain-consistent wrt $D_1, \ldots, D_n$, i.e. for all domains $D'_1, \ldots, D'_n$ such σ is domain-consistent wrt $D'_1, \ldots, D'_n$ we have $D'_1 \subseteq D_1 \& \ldots \& D'_n \subseteq D_n$.

Definition 3 A constraint store σ is *domain-satisfiable* iff none of its reduced domains is empty.

It is easy to show that the reduced domains of a constraint store σ exist and are unique and that all the solutions of σ are in its reduced domains. Domain consistency is thus a sound approximation of consistency.

Constraint entailment is replaced by the notion of domain entailment. Intuitively, a constraint is entailed by the constraint store if it is satisfied for all possible combinations of values that are still in the domains of the variables.

Definition 4 A constraint $c(x_1, \ldots, x_n)$ is *domain-entailed* by $D_1, \ldots, D_n$ iff, for all values $v_1, \ldots, v_n$ in $D_1, \ldots, D_n$, $c(v_1, \ldots, v_n)$ holds.

Definition 5 A constraint store σ *domain-entails* a constraint c iff c is domain-entailed by the reduced domains of σ.

Domain entailment is a sound relaxation of entailment: domain entailment implies entailment.

Finally generalization is replaced by the notion of domain generalization. Intuitively, the generalization of a set of constraint stores are range constraints obtained by taking the pointwise union of the reduced domains of the constraint stores.

Definition 6 The *generalization* of a set of constraint stores $\{\sigma_1, \ldots, \sigma_m\}$ is the constraint store

$$x_1 \in' \bigcup_{j=1}^{m} D_1^j \ \& \ \ldots \ \& \ x_n \in' \bigcup_{j=1}^{m} D_n^j$$

where $D_1^j, \ldots, D_n^j$ are the reduced domains of σ_j.

The definition of generalization satisfies property (1). It is not the strongest, but provides a practical compromise between efficiency and expressiveness.

3.1.5 Interval Reasoning

Interval reasoning is applied on the interval constraints $>'', \geq'', ='', \neq'', \leq'', <'', \in'', \notin''$. Instead of checking consistency of these constraints, `cc(FD)` enforces interval consistency. The basic difference compared to domain consistency is that the reasoning is only concerned with the minimum and maximum values in the domains. We now define the three operations for interval reasoning: interval consistency, interval entailment and interval generalization. In the following, we use D^* to denote the set $min(D)..max(D)$ where $min(D)$ and $max(D)$ denote respectively the minimum and maximum values in D.

Definition 7 A constraint c is *interval-consistent* wrt $D_1, \ldots, D_n$ if, for each variable x_i and value $v_i \in \{min(D_i), max(D_i)\}$, there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1^*, \ldots, D_n^*$ such that $c(v_1, \ldots, v_n)$ holds.

Note how only the lower and upper bounds are considered for variable x_i. The remaining definitions for interval satisfiability are modelled after those of domain satisfiability. Existence and uniqueness of the reduced domains for interval constraints can easily be shown as well as the soundness of interval satisfiability.

The first definition for interval entailment becomes as follows.

Definition 8 A constraint $c(x_1, \ldots, x_n)$ is *interval-entailed* by $D_1, \ldots, D_n$ iff, for all values $v_1, \ldots, v_n$ in $D_1^*, \ldots, D_n^*$, $c(v_1, \ldots, v_n)$ holds.

The remaining definitions are defined in a similar way as for domain-reasoning. Finally, the generalization is computed as follows.

Definition 9 The *generalization* of a set of constraint stores $\{\sigma_1, \ldots, \sigma_m\}$ is the constraint store

$$x_1 \in'' (\bigcup_{j=1}^{m} D_1^j)^* \; \& \; \ldots \; \& \; x_n \in'' (\bigcup_{j=1}^{m} D_n^j)^*$$

where $D_1^j, \ldots, D_n^j$ are the reduced domains of σ_j.

3.1.6 The Constraint Solver

Given a set of domain constraints S_d and interval constraints S_i, the constraint solver in `cc(FD)` checks if S_d and S_i are simultaneously domain-satisfiable and interval-satisfiable with respect to the same domains. It also reduces the domains accordingly.

Example 10 [Domain Consistency] The goal

```
?- X ∈′ 1..2, Y ∈′ 0..10, X =′ Y mod 3
```

produces the reduced domains $D_X = 1..2$ and $D_Y = \{1, 2, 4, 5, 7, 8, 10\}$. Adding the constraint `Y` $\notin$ `{2,5,8}` would produce the domains $D_X = \{1\}$ and $D_Y = \{1, 4, 7, 10\}$.

Example 11 [Interval Consistency] The goal

```
?- X ∈′ 1..2, Y ∈′ 0..10, X =″ Y mod 3
```

produces the reduced domains $D_X = 1..2$ and $D_Y = 1..10$. Adding the constraint `Y` $\notin$ `{2,5,8}` would produce the domains $D_X = 1..2$ and $D_Y = \{1, 3, 4, 6, 7, 9, 10\}$.

3.2 The Cardinality Combinator

Motivation The constraint solver in `cc(FD)` is only concerned with conjunction of constraints. Many practical applications however contain disjunctive information and an adequate processing of disjunctions is often a prerequisite to obtain a satisfactory solution. Consider, for instance, a disjunctive scheduling problem where two tasks i and j cannot be scheduled at the same time. The no-overlap constraint between these two tasks can be expressed as

```
disjunctive(Si,Di,Sj,Dj) :- Si + Di ≤″ Sj.
disjunctive(Si,Di,Sj,Dj) :- Sj + Dj ≤″ Si.
```

assuming that S_i, S_j are the starting dates of i and j and D_i,D_j their respective durations. The main problem with this formulation comes from the fact that the no-overlap constraint is only used for making choices and never to reduce the search space. However, when it is known that the constraint "task i precedes task j" is not consistent with the constraint store, the other alternative "task j precedes task i" must hold and hence can be added to the constraint store achieving early pruning of the search space. This handling of disjunctions requires constraint entailment as a primitive constraint operation and treats constraints locally. It enables the system to deduce constraints from disjunctions and is the key idea behind the cardinality operator which, in addition, generalizes this idea to threshold operators. The cardinality operator has been used in numerous applications including car-sequencing, disjunctive scheduling, hamiltonian path, and DSP scheduling to name a few.

Description In its most primitive form, the cardinality combinator is an expression of the form $\#(l, [c_1, \ldots, c_n], u)$ where l, u are integers and $c_1, \ldots, c_n$ are primitive constraints. Declaratively, it holds iff the number of true constraints in $[c_1, \ldots, c_n]$ is no less than l and no more than u. The cardinality operator generalizes the usual logical connectives. $c_1 \wedge \ldots \wedge c_n$ is equivalent to $\#(n, [c_1, \ldots, c_n], n)$, $c_1 \vee \ldots \vee c_n$ to $\#(1, [c_1, \ldots, c_n], n)$ and $\neg c$ to $\#(0, [c], 0)$. Other connectives can then be obtained easily.

The key feature of the cardinality combinator is its operational semantics. The main idea is that constraint entailment is used in a local manner to determine if the cardinality expression has a solution. When only one way of satisfying the cardinality is left, the appropriate constraints are added to the constraint store. More precisely, the two basic cases are:

1. a cardinality $\#(n, [c_1, \ldots, c_n], \_)$ requires $c_1, \ldots, c_n$ to be true; $c_1, \ldots, c_n$ are then added to the constraint store;

2. a cardinality $\#(\_, [c_1, \ldots, c_n], 0)$ requires $\neg c_1, \ldots, \neg c_n$ to be true; $\neg c_1, \ldots, \neg c_n$ are then added to the constraint store.

Assuming that σ is the constraint store at some computation step, the two reduction cases are:

- $\#(l, [c_1, \ldots, c_n], u)$ reduces to $\#(l-1, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n], u-1)$ if $\mathcal{C} \models (\forall)(\sigma \Rightarrow c_i)$;
- $\#(l, [c_1, \ldots, c_n], u)$ reduces to $\#(l, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n], u)$ if $\mathcal{C} \models (\forall)(\sigma \Rightarrow \neg c_i)$.

In practice, entailment is approximated through domain and interval entailment, depending on c_i. cc(FD) offers various extensions to the primitive form: l and u can be any arithmetic terms and the c_i can also be cardinality combinators. The last case is handled by means of a simple rewriting rule [32]. Logical connectives (prefixed with #) can also be used freely in cc(FD) and are interpreted as abbreviations for cardinality formulas. Finally, when only one bound is relevant, special forms such as `U #≥ [c1,...,cn]` and `L #≤ [c1,...,cn]` can be used. The implementation exploits the special forms to obtain better performance as discussed in the implementation section.

Example 12 [Disjunctive Constraints] The no-overlap constraint mentioned in the motivation can be expressed as

```
disjunction(Si,Di,Sj,Dj) :-
    1 #≤ [Si + Di ≤" Sj, Sj + Dj ≤" Si]).
```

It achieves the pruning described previously. When the negation of one of the constraints is implied by the constraint store, the other constraint is automatically added to the store. For instance, the goal

```
?- S₁ ∈′ 1..6, S₂ ∈′ 1..10, disjunction(S₁,7,S₂,6).
```

produces the reduced domain $S_1 \in' 1..3$, $S_2 \in' 8..10$. The no-overlap constraint is an important part of the disjunctive scheduling programs reported in the experimental results.

Example 13 [Communication Constraints] An interesting application of the cardinality combinator occurs in the Digital Signal Processing (DSP) application of [7, 31], whose results are also reported in the experimental results. The purpose of the application is to allocate tasks to processors in an architecture combining pipeline processing and master-slave processing in order to minimize the total delay of the DSP application. To solve the problem, it is necessary to express a communication constraint between each two successive tasks in the task graph of the application. The delay between two tasks is 0 when both tasks are assigned to the same processor, 1 when one of them is assigned to the master processor or if the processor of the second task follows the processor of the first task in the pipeline, and 2 otherwise (the communication goes through the master). It is expressed in `cc(FD)` by

```
delay(S₁,P₁,S₂,P₂) :-
      Delay ∈′ 0..2,
      Delay =′ 0 #⇔ P₁ =′ P₂,
      Delay =′ 1 #⇔ P₁ ≠′ P₂ #∧ (P₂ =′ P₁ + 1 #∨ P₁ =′ 1 #∨ P₂ =′ 1
      S₂ ≥′ S₁ + Delay.
```

In the above constraint, S_1, S_2 are the starting dates of tasks 1 and 2 and P_1, P_2 are their associated processors. The master processor is processor 1. This constraint is a key component of our solution which compares very well with a specific branch and bound algorithm written in C.

Example 14 [Capacity Constraints] The motivating example contains a third use of cardinality for the capacity constraints. The main technique here is to associating a boolean with a constraint using cardinality

```
B ∈′ 0..1, #(B,[c],B).
```

Arbitrary constraints on the boolean can now be expressed and two-way propagation takes place between the boolean and the constraint. This technique is used in the perfect square application.

3.3 Constructive Disjunction

Motivation Constructive disjunction was motivated by the need to achieve a more global pruning for disjunctions than the one offered by cardinality. Consider, for instance, the definition of `maximum(X,Y,Max)` which holds iff `Max` is the maximum of `X` and `Y`. Using cardinality, it can be expressed as

```
maximum(X,Y,Max) :-
   X ≤″ Max,
   Y ≤″ Max,
   Max =″ X #∨ Max =″ Y.
```

Unfortunately, the above implementation produces no pruning on the maximal value of `Max`. For instance, the goal

```
?- X ∈′ 5..10, Y ∈′ 4..11, Max ∈′ 0..20, maximum(X,Y,Max).
```

produces the reduced domains $D_X = 5..10, D_Y = 4..11, D_{Max} = 5..20$ because both constraints in the cardinality are treated locally and are consistent with the constraint store. Constructive disjunction makes sure to produce $D_{Max} = 5..11$.

Description A constructive disjunction is an expression of the form $\sigma_1 \vee' \ldots \vee' \sigma_n$ or $\sigma_1 \vee'' \ldots \vee'' \sigma_n$. The difference between $\vee'$ and $\vee''$ comes from the two forms of generalizations available in `cc(FD)`: domain generalization and interval generalization. `cc(FD)` allows also the presence of cardinality formulas and constructive disjunctions in the disjuncts by using simple rewriting rules.

Declaratively, a constructive disjunction can be read as a simple disjunction. The operational behaviour is however the important feature. If any of the disjuncts is entailed by the current constraint store σ, then the constructive disjunction is clearly satisfied. Otherwise, the new constraint store is simply $\sigma \wedge \Gamma$ where Γ is the domain or interval generalization of $\{\sigma \wedge \sigma_1, \ldots, \sigma \wedge \sigma_n\}$.

Of course, the generalization is computed incrementally (and added to the constraint store) each time the constraint store is modified. In other words, the idea is to extract, at any computation step, common information from the disjuncts in conjunction with the constraint store. In `cc(FD)`, the common information takes the form of range constraints.

Example 15 [Maximum Constraints] The maximum constraint can now be expressed as

```
maximum(X,Y,Max) :-
   X ≤″ Max,
   Y ≤″ Max,
   Max =″ X ∨″ Max =″ Y.
```

The goal

```
?- X ∈′ 5..10, Y ∈′ 4..11, Max ∈′ 0..20, maximum(X,Y,Max).
```

leads to the reduced domains $D_{Max} = 5..10, D_X = 5..10$ for the first disjuncts and to $D_{Max} = 5..11, D_Y = 5..11$ for the second disjunction. The generalization produces the domains $D_{Max} = 5..11, D_X = 5..10, D_Y = 4..11$. The maximum constraint is an important component of the solution to disjunctive scheduling problems.

Example 16 [Distance Constraints] Another example of constructive disjunction is the handling of constraints of the form $|X - Y| \geq I$. This is used in the applications referred to as `satel1` and `satel2` in the experimental results. The implementation is simply

```
absolute_distance(X,Y,I) :-
   X - Y ≥′ I ∨′ Y - X ≥′ Y.
```

Contrary to the `maximum` constraint which only makes pruning on the bounds of the domains, the above constraint removes values in the middle of the domains. For instance, the query

```
?- X ∈′ 1..10, Y ∈′ 1..10, absolute_distance(X,Y,8).
```

produces the reduced domains $D_x = \{1,2,9,10\}, D_y = \{1,2,9,10\}$.

Example 17 [Disjunctive Scheduling] In the previous examples, the disjuncts were simple primitive constraints but in `cc(FD)` they can be any constraint store, i.e. any conjunction of primitive constraints. For instance, in disjunctive scheduling, one often need conditional expressions of the form

$$\texttt{(Min} \geq'' \texttt{X}_1 \texttt{ , X}_1 =' \texttt{Entry)} \vee'' \ldots \vee'' \texttt{(Min} \geq'' \texttt{X}_n \texttt{ , X}_n =' \texttt{Entry)}.$$

Operationally, the intention is that `Min` be greater than at least one of the $\texttt{X}_i$ that can be equal to `Entry`.

3.4 The Implication Combinator

Motivation Blocking implication [21, 27, 15] is a combinator generalizing coroutining mechanisms in logic programming. The main idea behind coroutining mechanisms is to postpone execution of a goal until some conditions on its variables are satisfied. The main idea behind blocking implication is to use constraints for the conditions. As a consequence, blocking implication is a convenient tool to implement local propagation of values, pruning rules, and algorithm animation. It is used in many applications including hamiltonian circuits, test generation, and disjunctive scheduling. All graphical animations also use blocking implication.

Description In its simplest form, a blocking implication is an expression of the form $c \rightarrow B$ where c is a primitive constraint and B is a body. Declaratively, it can be read as an implication. The key feature is once again the operational semantics. The body of the implication is executed only if c is entailed by the constraint store. If $\neg c$ is entailed by the constraint store, the implication simply succeeds. Otherwise, the implication suspends and the body will be executed only when a latter constraint store entails c due to the addition of other constraints.

`cc(FD)` also allows cardinality formulas instead of the constraints since once again the operational semantics can be given by simple rewrite rules. It also allows expressions such as `fixed(T)` with `T` being an arithmetic term to be used instead of c. An expression `fixed(T)` $\rightarrow$ `B` executes `B` as soon as `T` is constrained to take a unique value by the constraint store and is an abbreviation of the constraint `#(1, [T` $='$ *min_int*`, T` $='$ *min_int*$+1$`, ..., T` $='$ *max_int* `], 1)` where *min_int* and *max_int* are a lower and upper bound of the finite set of natural numbers that can fit in a memory word.

Example 18 [Local Propagation] Local propagation can be implemented in a simple way using blocking implication. For instance, a logical and-gate using local propagation techniques would be:

```
and(X,Y,Z) :-
     X =' 0 → Z =' 0,
     Y =' 0 → Z =' 0,
     Z =' 1 → ( X =' 1 , Y =' 1 ),
     X =' 1 → Y =' Z,
     Y =' 1 → X =' Z,
     X =' Y → X =' Z.
```

The first rule says that, as soon as the constraint store entails `X = 0`, the constraint `Z = 0` must be added to the constraint store. Note that the last three rules which actually do more than local value propagation; they also propagate symbolic equations and one of them is conditional to a symbolic equality.

Example 19 [Disjunctive Scheduling] In disjunctive scheduling, a number of tasks are required not to overlap. A typical pruning technique amounts to establishing which tasks can be entry of the disjunction (i.e. can be scheduled first) and which tasks can be exit of the disjunction (i.e. can be scheduled last). To determine the entry, a typical rule is

$$\mathtt{S}_i \ \mathtt{+\ TotalDuration} >' \mathtt{ExitDate} \rightarrow \mathtt{Entry} \neq' i$$

where $\mathtt{S}_i$ represents the starting date of a task, `TotalDuration` the summation of the durations of all tasks in the disjunction, and `ExitDate` is the maximum end date of the tasks which can be exits of the disjunction. It simply expresses that if the constraint store implies that the starting date of task i added to the total duration is greater than the maximum end date, then task i cannot be an entry of the disjunction.

Example 20 [Algorithm Animation] Blocking implication is the main tool to produce graphical algorithm animation. For instance, in a n-queens problem, the animation would show the queens already placed and the values removed from the remaining queens. The animation is obtained by using blocking implications of the form

$$\mathtt{fixed(Q2)} \rightarrow \mathtt{show\_queens(Q2,2)}$$

to display the queen associated with column 2 and

$$\mathtt{Q2} \neq' \mathtt{4} \rightarrow \mathtt{show\_removed(4,2)}$$

to show that the value 4 is no longer possible for queens 2. The appeal of this approach is that the graphical animation is completely separated from the program and runs in coroutining.

3.5 Higher-Order Predicates

`cc(FD)` contains also a number of higher-order predicate for optimization purposes. The basic forms are

```
minof(Goal,Function,Res)
maxof(Goal,Function,Res)
minof_r(Goal,Function,Res)
maxof_r(Goal,Function,Res)
```

The purpose of these predicates is to obtain an optimal solution to a goal with respect to an objective function (i.e. an arithmetic term). Two versions of the predicates are given. The first version uses a depth-first branch and bound algorithm while the second version uses a restarting strategy. Special care is taken in the depth-first branch and bound when a new solution is found to backtrack to a point where the solution can potentially be improved upon. The restarting strategy may be of interest when heuristics are strongly influenced by the value of the best solution found so far [26]. The typical technique to solve optimization problems amounts to embedding the choice part in the higher-order predicate:

```
solve_problem(...) :-
     create_variables(...),
     state_constraints(...),
     minof(make_choice(...),Function,Res).
```

Finally, `cc(FD)` also contains a number of non-logical predicates giving access to the domains. These predicates should only be used for defining heuristics in the choice process (e.g. choosing the next variable to instantiate as the one with the smallest domain).

4 Implementation

As mentioned previously, the implementation of `cc(FD)` includes a version of the WAM [36], suitably enhanced with constraint processing facilities. The introduction of constraints is almost exclusively achieved by a set of built-in predicates, keeping the interface between the two parts to a strict minimum. In particular, no new instructions have been added to WAM apart from those necessary to achieve the coroutining facilities required by the implication combinator. As a consequence, `cc(FD)` preserves the simplicity and speed of the WAM. Note also that the implementation does not sacrifice the efficiency of constraint solving as our experimental results indicate. The specialization of constraints simply does not occur at the WAM level but inside the constraint solver. In the rest of this section, we concentrate on the main features of the constraint system and of the combinators.

4.1 Constraint System

Domain representation is an important aspect of the constraint system. `cc(FD)` uses different domain representations depending on the application. When the implementation only needs interval reasoning for a given domain variable, the domain representation is simply two integers: a lower and an upper bound. When some values are removed from the middle of the domain, a more explicit representation, an array of booleans, is constructed to indicate the presence or the absence of the element. This is completely transparent to the user.

Constraints are attached directly to parts of the domain representation. For instance, inequalities are attached to the lower and upper bounds of the domains; $\texttt{X} \leq' \texttt{Y}$ is attached to the lower bound of `X` (to update the lower bound of `Y`) and to the upper bound of `Y` (to update the upper bound of `X`). Disequations are attached to the domain as a whole and are only considered when one variable is instantiated. Finally, constraint entailment (e.g. entailment of $\texttt{X} \neq' \texttt{3}$) also attaches constraints to elements of the boolean array. This enables the system to check entailment of unary constraints (a very frequent case) in constant time over the whole execution. Once again, the representation is adapted depending on the need of the application.

Modifications to the domains are trailed by remember pairs $< address, oldvalue >$. Time stamps are used to avoid trailing twice the same address in between two choice points.

4.2 Constraint Algorithms

Constraints are also classified depending upon their complexity. `cc(FD)` has specialized algorithms for nonlinear, linear, binary, and unary constraints. Once again, this is fully transparent to the user. The specialization is performed at run-time in the present implementation but global flow analysis should allow us to move most of the work at compile-time in the next version of the system.

The constraint-solving algorithms are based on (non-binary) generalization of the AC-5 algorithm [34] using a breath-first strategy. In particular, domain-consistency of any combination of binary functional (e.g. $\mathtt{X} =' \mathtt{Y}$), anti-functional (e.g. $\mathtt{X} \neq' \mathtt{Y}$), monotonic (e.g. $\mathtt{X} >' \mathtt{Y}$), and piecewise constraints (e.g. $\mathtt{X} =' \mathtt{Y}\ \mathtt{mod}\ 7$), require $O(cd)$ amortized time, where c is the number of constraints and d is the size of the largest domain [34]. Once again, the system (dynamically) compiles constraints differently depending on their properties. For instance, a constraint such as $\mathtt{X} =' \mathtt{Y}\ \mathtt{mod}\ \mathtt{c}$ will be compiled into expressions of the form

$$\forall\ \alpha:\quad \mathtt{Y} \neq \alpha \rightarrow \mathtt{X} \neq \alpha\ \mathtt{mod}\ \mathtt{c}$$
$$\forall\ \alpha:\quad \mathtt{X} \neq \alpha \rightarrow \mathtt{Y} \neq \mathtt{N} * \mathtt{C} + \alpha$$

when the constraint is recognized as functional due to the domains of the variables. Operationally these expressions can be seen as abbreviations for a finite number of blocking implications. The implementation however uses constant space to represent them. When the above constraint is not functional, it behaves operationally as a set of cardinality formulas of the form

$$\mathtt{X} =' \alpha\ \#\Leftrightarrow\ \mathtt{Y} \in' \mathtt{S}$$

where S is the set of values supporting α. At the implementation level, the space requirement is proportional, not the size of the domains, but rather to the number of groups in the piecewise decompositions.

Interval consistency of non-binary monotonic constraints requires $O(cdn^2)$ amortized where n is the number of variables in the largest constraints. An optimal algorithm of complexity $O(cdn)$ exists [33] for linear constraints but our preliminary experimentations indicate that its overhead may reduce its interest.

The breath-first strategy makes sure that domain consistency of monotonic constraints has a complexity which is quadratic in the number of variables and constraints independently of the domain sizes contrary to a depth-search strategy which may be exponential.

4.3 The Cardinality Operator

A cardinality operator of the form $\#(l, [c_1, \ldots, c_n], u)$ is implemented by keeping two counters for the number of formulas which are true and false respectively. In addition, the system spawns n constraint-entailment procedures checking if the c_i or their negations are entailed by the constraint store. When the true-counter reaches the upper bound, all remaining constraints are forced to false, i.e. their negations are added to the constraint store. When the false-counter reaches $n - l$, all remaining constraints are forced to be true, i.e. they are added to the constraint store. Specific optimizations are possible for various specialized forms. For instance, when the lower bound is unimportant (e.g. $u\ \#\geq\ [c_1, \ldots, c_n]$), entailment needs only to be checked for the constraints $c_1, \ldots, c_n$ and not their negations.

Note also that our implementation of cardinality enables to implement arc-consistency on arbitrary binary constraints within the optimal (time and space) bounds of the AC-4 algorithm [22].

4.4 Constructive Disjunction

Constructive disjunction in cc(FD) is implemented in terms of constraint solving in order to obtain an incremental behaviour. The key idea is to rename the variables in each disjunct independently and to add the renamed disjuncts to the constraint store. In doing so, the implementation reuses the algorithms available for constraint solving, achieving both efficiency and reuse of existent code. The astute reader would have noticed that special care is needed in case of failures. The connections between the renamed variables and the original variables is achieved through a number of (internal) constraints which are essentially of two types

1. **subsumption constraints:** these constraints force the domain of a variable to be a subset of the domain of another variable;
2. **union constraints:** these constraints force the domain of a variable to be a subset of the union of the domains of other variables.

Subsumption constraints have been investigated previously by Parker [24] as a language extension. In `cc(FD)`, they are only used inside the implementation since their directed nature is somewhat in contradiction with the multi-directional philosophy of constraint logic programming. Many optimizations are present in the system to handle efficiently the cases where some variables appear only in a subset of the disjuncts. For instance, these optimizations make sure that constructive disjunction comes close to cardinality for the case where both apply and constructive disjunction does not produce more pruning.

4.5 Blocking Implication

Blocking implementation is simply a generalization of the traditional `if-then-else` construct. The compilation schema is simply

```
< check entailment of the constraint >
JUMPIFNOTTRUE labelfalse
     < execute body >
     JUMP next
labelfalse:  JUMPIFFALSE next
     < handle suspension >
next:
```

The handling of suspension amounts to creating an entailment procedure for the constraint and attaching the body to the procedure. Whenever a constraint is entailed, its associated body is inserted in a list of bodies which are executed as soon as possible, i.e. after a built-in procedure or at the neck of a user-defined procedure. The list is executed in a depth-first manner for simplicity and closely follows the traditional implementation of delay mechanisms (e.g. [6]).

5 Experimental Results

In this section, we report a number of experimental results of `cc(FD)`. All times are for a Sun Sparc Station I (Sun 4/60). Table 1 shows the search time, the total time, the potential search space, the number of variables, and the number of constraints for a number of problems, and the number of lines of the program. The search time is the time spent in the nondeterministic part of the program while the total time includes reading of data, creating the variables, and stating the constraints. The number of variables and constraints are taken just before the nondeterministic part of the program although, in some cases, constraints are generated during the choice process as well. The potential search space does not always reflect the difficulty of the problem but should provide some more indication on the sizes of the problems dealt with by `cc(FD)`. The number of lines (which includes blank lines and comments) gives also an idea of the compactness of the programs which enables a short development time. `Bridge` is a disjunctive scheduling problem from [2], `car` is a car-sequencing problem [?, 25], `cutting` is the numerical statement of a cutting-stock problem taken from [10], `satel1, satel2` are two resource allocation problems with distance constraints, `square` is the perfect packing problem, `hamilton` is the Euler knight problem, `donald, sendmory` are two cryptarithmetic problems, `queens8, queensall, queens96` are n-queens programs to find respectively the first solution to the 8-queens problem, all solutions to the 8-queens problem, and the first solution to the 96 queens problems. `magic11, magic16, magic21` are various instances of the magic series problem taken from [32, 30] for sizes 11, 16, and 21.

Table 2 compares `cc(FD)` with a specialized branch and bound algorithm written in C on a number of DSP problems [7, 19]. Both algorithms were run on the same machine. As can be seen from the data, `cc(FD)` compares very well with the specialized program especially for the largest problems. The `cc(FD)` program is about 200 lines long.

Table 3 compares `cc(FD)` with a specialized scheduling algorithm [4]. These are very difficult scheduling problems, requiring sophisticated handling of disjunctions. The potential search space of a 6/10 problem is 2^{330}. `cc(FD)`, in its present state, cannot compete in pure speed with the specialized program but the difference is mainly a constant factor, showing that the pruning techniques of `cc(FD)` are quite effective. The `cc(FD)` program is about 440 lines long. Finally, it is interesting to point out that the `cc(FD)` program is able to solve optimally and prove optimality of a famous 10/10 job shop scheduling which was posed in 1963 [23] and left open for 25 years before being solved in [5]. The algorithm in [5] is very involved including relaxation techniques to preemptive scheduling. This problem requires about 90 hours of computation.

The above results seem to indicate that `cc(FD)` is a step in closing the gap between declarative constraint languages and procedural languages. Very difficult problems are now in the scope of `cc(FD)`, which comes close in efficiency to specialized algorithms written in procedural programs.

6 Conclusion

In this paper, we have presented the design, implementation, and applications of `cc(FD)`, a declarative nondeterministic constraint language over finite domains. `cc(FD)` is a small and uniform language based on a conceptually simple constraint solver and a number of general-purpose combinators. The key novelty in `cc(FD)` is the availability of the combinators which enable to address, at the language level, issues such as the handling of disjunctions, the definition of non-primitive constraints, and the control of the search exploration. The implementation of `cc(FD)` (about 40,000 lines of C) includes optimal consistency algorithms, adaptable data structures, and incremental techniques for the combinators. The experimental results indicate that `cc(FD)` can tackle very difficult problems with an efficiency which comes close to procedural languages in many cases. Future work on `cc(FD)` will be devoted to the generalizations of the combinators to arbitrary goals and to global flow analysis to specialize constraints and data structures at compile time. These extensions may further improve expressiveness and efficiency. Finally, instantiations of the framework to Boolean algebra and rational numbers are currently developed.

Acknowledgements

Discussions with Alain Colmerauer and Michel Van Caneghem help simplifying the design and solving the motivating example. This research was partly supported by the National Science Foundation under grant number CCR-9108032 and the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225.

References

[1] A. Aggoun and N. Beldiceanu. Extending CHIP To Solve Complex Scheduling and Packing Problems. In *Journées Francophones De Programmation Logique*, Lille, France, 1992.

[2] M. Bartusch. *Optimierung von Netzplaenen mit Anordnungsbeziehungen bei Knappen Betriebsmitteln.* PhD thesis, Fakultaet fur Mathematik und Informatik, Universitaet Passau (F.R.G), 1983.

[3] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[4] J. Carlier. Ordonnancement A Constraintes Disjonctives. *RAIRO Operations Research*, 12(4):333–351, November 1978.

[5] J. Carlier and E. Pinson. Une Méthode Arborescente pour Optimiser la Durée d'un JOB-SHOP. Technical Report ISSN 0294-2755, I.M.A, 1986.

[6] M. Carlsson. Freeze, Indexing and Other Implemenation Issues on the WAM. In J-L. Lassez, editor, *Fourth International Conference on Logic Programming*, pages 40–58, Melbourne, Australia, 1987.

[7] J.W. Chinneck, R.A. Goubran, G.M. Karam, and M Lavoie. A Design Approach For Real-Time Multiprocessor DSP Applications. Report sce-90-05, Carleton University, Ottawa, Canada, February 1990.

[8] N. Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press, New York, 1975.

[9] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[10] M.C. Costa. Une etude pratique de decoupes de panneaux de bois. *RAIRO Recherche Operationnelle*, 18(3):211–219, August 1984.

[11] T. Dean and M. Boddy. An Analysis of Time-dependent Planning. In *Proceedings of the Seventh National Conference On Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, August 1988.

[12] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[13] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[14] T. Graf. Extending Constraint Handling in Logic Programming to Rational Arithmetic. Internal Report, ECRC, Munich, Septembre 1987.

[15] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. In *International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August 1989.

[16] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[17] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[18] M. Kubale and D. Jackowski. A Generalized Implicit Enumeration Algorithm for Graph Coloring. *CACM*, 28(4):412–418, 1985.

[19] Marco Lavoie. Task Assignment In A DSP Multiprocessor Environment. Master Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, August 1990.

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

[21] M.J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[22] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[23] J.F. Muth and G.L. Thompson. *Industrial Scheduling*. Prentice Hall, Englewood Cliffs, NJ, 1963.

[24] R.G. Parker and R.L. Rardin. *Discrete Optimization*. Academic Press, London (England), 1988.

[25] B.D. Parrello. CAR WARS: The (Almost) Birth of an Expert System. *AI Expert*, 3(1):60–64, January 1988.

[26] H.M. Salkin. On the Merit of the Generalized Origin and Restarts in Implicit Enumeration. *Opns. Res.*, 18:549–554, 1970.

[27] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.

[28] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[29] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of Ninth ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[30] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[31] P. Van Hentenryck. Scheduling and Packing in the Constraint Language `cc(FD)`. Technical Report CS-92-43, CS Department, Brown University, 1992.

[32] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A New Logical Connective and Its Application to Constraint Logic Programming. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991. Also to appear in *Constraint Logic Programming: Selected Research*, The MIT Press, 1992.

[33] P. Van Hentenryck, Y. Deville, M.L. Chen, and C.M. Teng. New Results in Consistency of Networks. Technical report, CS Department, Brown University, 1992. Forthcoming.

[34] P. Van Hentenryck, Y. Deville, and C.M. Teng. A Generic Arc Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57(2-3), 1992.

[35] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 5(2-4), 1992.

[36] D.H.D Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, October 1983.

Problem	Search Time	Total Time	Search Space	Variables	Constraints	lines
Bridge	3.9	4.6	2^{77}	46	445	140
Car	0.92	9.37	20^{100}	600	12390	225
Cutting	7.8	11.3	4^{72}	72	79	303
Satel1	9.8	41.6	24^{200}	5158	6678	338
Satel2	8.1	13.09	$44^{98}36^{102}$	1362	2911	338
square	38.15	60.66	21^{224}	9366	52584	105
hamilton	1.45	4.61	$2^{92}3^{40}$	64	6560	166
donald	0.05	0.06	10^{10}	15	63	50
sendmory	0.00	0.01	8^{10}	8	38	46
queens8	0.02	0.04	8^{8}	8	92	52
queens8all	0.63	0.65	8^{8}	8	92	52
queens96	0.80	2.94	96^{96}	96	13776	52
magic11	0.14	0.25	11^{11}	11	165	58
magic16	0.39	0.57	16^{16}	16	320	58
magic21	0.77	1.12	21^{21}	21	525	58

Table 1: Experimental Results of cc(FD)

Problem	Size	Processors	Topology	Total Delay	cc(FD)	Specialized BB
RDAD01	9	3	Pipeline	3	0.78	0.016
RDAD02	9	3	Pipeline	3	0.72	0.016
RDAD03	6	5	Architecture-like	5	0.22	0.000
RDAD04	19	6	Parallel Pipelines	3	1.66	51.700
RDAD05	12	4	Pipeline	3	1.12	0.016
RDAD06	16	5	Parallel Pipelines	3	2.68	6.300
RDAD07	12	4	Parallel Pipelines	2	0.56	0.050
RDAD08	15	5	Merging Tasks	3	3.23	963.13
RDAD09	9	6	Many Generators	2	0.18	0.016
RDAD10	15	5	Parallel Pipelines	4	3.90	0.033
RDAD20	13	5	Architecture-like	3	0.99	0.016
RDAD40	25	8	Parallel Pipelines	5	54.40	?????
RDAD41	25	8	Parallel Pipelines	4	4.24	0.100

Table 2: Results on Actual DSP Applications

Nb. of Machines	Nb. of jobs	Nb. of tasks	cc(FD)	Specialized BB
5	11	55	26	4
4	13	52	27	2
5	12	60	11	7
4	14	56	81	24
6	10	60	620	158
9	8	56	578	209
7	7	49	246	37

Table 3: Results on Disjunctive Scheduling Application