Scheduling and Packing In The Constraint Language cc(FD)

Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-43
September 1992

Scheduling and Packing In The Constraint Language cc(FD)

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

Abstract

Constraint Logic Programming (CLP), and its generalization in the cc framework, define a class of declarative constraint languages combining nondeterministic goal-directed programming with constraint techniques over an arbitrary domain. CLP languages are particularly attractive for combinatorial search problems as they offer a short development time and a reasonable efficiency. In this paper, we present the application of cc(FD), a CLP language using consistency techniques over finite domains, to two applications: the perfect square problem and a Digital Signal Processing (DSP) problem. The perfect square problem amounts to placing squares of different sizes in a master square in an exact manner (i.e. no empty space remains). We show a natural and very short cc(FD) program solving the problem for 21 and 24 squares in a couple of seconds. The DSP application amounts to scheduling tasks on a multiprocessor in presence of precedence, capacity, and delay constraints. The complexity of the problem is exacerbated by the non-uniform communication delays coming from the architecture which combines pipeline and master-slave processing. We present a short cc(FD) program which compares very well with a special-purpose branch and bound algorithm. Both applications show the versatility of languages such as cc(FD) for the solving of discrete combinatorial search problems.

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 [12]. 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 [23], Booleans [2] and rational numbers [11, 26], Prolog III [6] is endowed with constraint systems over Booleans, rational numbers, and lists, while CLP($\Re$) [13] solves constraints over real numbers. The CLP scheme was further generalized into the cc framework of concurrent constraint programming [19, 20, 21] to accommodate additional constraint operations (e.g. constraint entailment [17]) and new ways of combining them (e.g. implication or blocking ask [19] and cardinality [24]).

CLP languages[1] support, in a declarative way, the solving of combinatorial search prob-

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

lems 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 [3], graph coloring [14], Hamiltonian circuits [5], microcode labeling [10]) and provides, at the same time, the basis for exact methods as well as approximate solutions (giving rise to the so-called "anytime algorithms" [7]).

The purpose of this paper is to demonstrate the use of CLP languages for solving discrete combinatorial search problems such as assignment, scheduling and packing problems by showing the solutions of two problems.

The CLP language used in the above problems is `cc(FD)`, an instance of the `cc` framework over finite domains that is best seen as a successor to the finite-domain part of `CHIP`. Both languages support the use of consistency techniques and local propagation in conjunction with don't-know nondeterminism approximated by backtracking. In addition, they support depth-first branch and bound for combinatorial optimization problems. The novel aspects of `cc(FD)` include the definition of new general-purpose combinators (such as cardinality, implication, constructive disjunction, and indexical constraints) and the availability of constraint entailment and constraint generalization as primitive operations on constraints. `cc(FD)` generalizes in an elegant way (and thus makes unnecessary) several features and constraints of `CHIP` that were difficult to justify theoretically. As a consequence, it provides additional operational expressiveness, flexibility, and efficiency and let us tackle problems such as disjunctions of constraints and the definition of primitive constraints.

The two applications chosen to illustrate the language are the perfect square problem and a Digital Signal Processing (DSP) application. The perfect square problem amounts to packing a number of small squares in a larger square without leaving any empty space. We show a rather short and natural `cc(FD)` program packing 21 or 24 squares in about 30 seconds on a SUN 4/60. The DSP application consists in scheduling tasks on a number of processors in order to minimize the total delay of the DSP function. The complexity of the DSP application comes from the various types of constraints, including precedence constraints, capacity constraints, and non-uniform communication delays. We show a `cc(FD)` program which is about 4 pages long and compares well with a specific branch and bound algorithm especially designed for the task. Other applications of CLP languages over finite domains to similar problems can be found in [9, 8, 10, 23, 22].

The rest of the paper is organized as follows. The next section gives an overview of the features of `cc(FD)` used in the applications. Each feature is illustrated on scheduling examples and provide the necessary background for the two applications. The next two sections describe the applications. The last section contains the conclusion of the paper.

```
Program  ::=  Clauses
Clauses  ::=  Head :- Body | Clauses Clauses
Head     ::=  Atom
Goal     ::=  Atom
Body     ::=  true | Goal | c | Body , Body | #(l,u,[c1,...,cn])
```

Figure 1: An Outline of the Syntax

```
p(X,Y,X) :-
   X ∈ 0..10, Y ∈ 0..10, Z ∈ 0..10,
   X ≥ Z + 3,
   Y ≤ Z,
   q(X,Y,Z).

q(X,Y,Z) :-
   r(X,Y).
q(X,Y,Z) :-
   Z ≥ Y + 2.

r(X,Y) :-
   X ≤ Y + 2.
```

Figure 2: A Simple Program

2 Overview of cc(FD)

Here we give an informal overview of the relevant parts of cc(FD). Section 2.1 sketches the syntax of the language, Section 2.2 introduces the CLP scheme, Sections 2.3, 2.4, and 2.5 discuss the details of constraint solving in cc(FD), the cardinality combinator and the meta-level predicates for optimization.

2.1 Syntax

Figure 1 shows an outline of the syntax of cc(FD) programs, as relevant to this paper. A cc(FD) is a set of clauses in which each clause has a head and a body. A head is an atom, i.e. an expression of the form $\mathtt{p(t_1,\ldots,t_n)}$ where $\mathtt{t_1,\ldots,t_n}$ are terms. A term is a variable (e.g. X) or a function symbol of arity n applied to n terms (e.g. f(X,g(Y))). A body is either `true` (the empty body), a goal (procedure call), a constraint (constraint solving), or a cardinality combinator. In this paper, variables are denoted by uppercase letters, constraints by the letter c, conjunctions of constraints by the letter σ, terms by letters t, s, atoms by letters H, B, goals by the letter G, and integers by the letters l, u, v, all possibly subscripted or superscripted. We also use $\mathcal{C}$ to denote a constraint system and D, possibly subscripted, to denote a finite domain. To illustrate the operational semantics of (part of) cc(FD), we use the simple program depicted in Figure 2.

2.2 The CLP Scheme

At least from a conceptual standpoint, the operational semantics of the CLP scheme is a simple generalization of the semantics of logic programming. It can be described as a goal-directed derivation procedure from the initial goal using the program clauses. A *computation state* is best described by

1. *a goal part*: the conjunction of goals to be solved;

2. *a constraint store*: the set of constraints accumulated so far.

Initially the constraint store is empty and the goal part is the initial goal. In the following, we denote the computation state by pairs $\langle\ G \ \Box\ \sigma\ \rangle$, where G is the goal part and σ is the constraint store. We use ϵ to denote an empty goal part or constraint store. An example computation state is

$\langle$ q(X,Y,Z) $\Box$ X,Y,Z $\in \{0,\ldots,10\}$ & X$\geq$Z+3 & Y$\leq$Z $\rangle$.

A *computation step* (i.e. the transition from one computation state to another) can be of two types depending upon the selection of an atom or a constraint in the goal part. In the first case, a computation step amounts to

1. selecting an atom in the goal part;

2. finding a clause that can be used to resolve the atom; this clause must have the same predicate symbol as the atom and the equality constraints between the goal and head arguments must be consistent with the constraint store;

3. defining the new computation state as the old one where the selected atom has been replaced by the body of the clause and the equality constraints have been added to the constraint store.

In the second case, a computation step amounts to

1. selecting a constraint in the goal part that can be satisfied with the constraint store;

2. defining the new computation state as the old one where the selected constraint has been removed from the goal part and added to the constraint store.

For instance, given a computation state

$\langle$ q(X,Y,Z) $\Box$ X,Y,Z $\in \{0,\ldots,10\}$ & X$\geq$Z+3 & Y$\leq$Z $\rangle$

a computation step can be performed using clause 2 of q to obtain a new computation state

$\langle$ Z$\geq$Y+2 $\Box$ X,Y,Z $\in \{0,\ldots,10\}$ & X$\geq$Z+3 & Y$\leq$Z $\rangle$.

Another computation step leads to the configuration

$\langle$ ϵ $\Box$ X,Y,Z $\in \{0,\ldots,10\}$ & X$\geq$Z+3 & Y$\leq$Z & Z$\geq$Y+2 $\rangle$, since the resulting constraint store is satisfiable. Note that, strictly speaking, equations should have appeared between the variables in the above example; they were omitted for clarity, since the variables have the same names in the program.

As should be clear, the basic operation of the language amounts to deciding the satisfiability of a conjunction of constraints. Note also that each computation state has a satisfiable constraint store. This property is exploited inside CLP languages to avoid solving the satisfiability problem from scratch at each step. Instead, CLP languages keep a reduced (e.g. solved) form of the constraints and transform the existing solution into a solution including the new constraints. Hence the constraint solver is made incremental. For instance, the last constraint store may be represented as

$$\langle\ \epsilon \ \square\ \mathtt{X} \in \{5, \ldots, 10\}\ \&\ \mathtt{Y} \in \{0, \ldots, 5\}\ \&\ \mathtt{Z} \in \{2, \ldots, 7\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \&\ \mathtt{Z}{\geq}\mathtt{Y}{+}2\ \rangle.$$

A computation state is *terminal* if

- the goal part is empty;
- no clause can be applied to the selected atom to produce a new computation state or the selected constraint cannot be satisfied with the constraint store.

A *computation* is simply a sequence of computation steps that either ends in a terminal computation state or diverges. A finite computation is *successful* if the final computation state has an empty goal, and *fails* otherwise.

To illustrate computations in a CLP language, consider our simple program again. The program has only one successful computation, namely

$\langle\ \mathtt{p(X,Y,Z)} \ \square\ \epsilon\ \rangle$
$\qquad\qquad\downarrow$ `(selecting the first constraint)`
...
$\qquad\qquad\downarrow$ `(selecting the last constraint)`
$\langle\ \mathtt{q(X,Y,Z)} \ \square\ \mathtt{X,Y,Z} \in \{0, \ldots, 10\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \rangle$
$\qquad\qquad\downarrow$ `(using clause 2 of` q`)`
$\langle\ \mathtt{Z}{\geq}\mathtt{Y}{+}2 \ \square\ \mathtt{X,Y,Z} \in \{0, \ldots, 10\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \rangle$
$\qquad\qquad\downarrow$ `(selecting the constraint)`
$\langle\ \epsilon \ \square\ \mathtt{X,Y,Z} \in \{0, \ldots, 10\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \&\ \mathtt{Z}{\geq}\mathtt{Y}{+}2\ \rangle$

The program has also one failed computation:

$\langle\ \mathtt{p(X,Y,Z)} \ \square\ \epsilon\ \rangle$
$\qquad\qquad\downarrow$ `(selecting the first constraint)`
...
$\qquad\qquad\downarrow$ `(selecting the last constraint)`
$\langle\ \mathtt{q(X,Y,Z)} \ \square\ \mathtt{X,Y,Z}\ \in \{0, \ldots, 10\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \rangle$
$\qquad\qquad\downarrow$ `(using clause 1 of` q`)`
$\langle\ \mathtt{r(X,Y,Z)} \ \square\ \mathtt{X,Y,Z}\ \in \{0, \ldots, 10\}\ \&\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \rangle$
$\qquad\qquad\downarrow$ `(using clause 1 of` r`)`
$\langle\ \mathtt{X}{\leq}\mathtt{Y}{+}2 \ \square\ \mathtt{X,Y,Z}\ \in \{0, \ldots, 10\}\ \&\ \mathtt{X}{\geq}\mathtt{Z}{+}3\ \&\ \mathtt{Y}{\leq}\mathtt{Z}\ \rangle$

The last computation state is terminal since the conjunction of constraints

$\mathtt{X} \geq \mathtt{Z}+3$ & $\mathtt{Y} \leq \mathtt{Z}$ & $\mathtt{X} \leq \mathtt{Y}+2$

is not satisfiable.

Note that the results of the computation are the constraint stores of the successful computations. Also, nothing has been said so far on the strategy used to explore the space of computations. Most CLP languages use a computation model similar to Prolog: atoms are selected from left to right in the clauses, clauses are tried in textual order, and the search space is explored in a depth-first manner with chronological backtracking in case of failures.[2] For instance, on the simple program, a CLP language typically uses clause 1 for `p`, then clause 1 for `q`, and finally encounters a failure when trying to solve `r`. Execution then backtracks to clause 2 of `q`, giving the successful computation.

2.3 The Constraint Solver

In this section, we consider the finite domain part of `cc(FD)` in isolation (i.e. without discussing first-order terms). `cc(FD)` works on variables ranging over a finite set of natural numbers (e.g. positive integers that can fit in a memory word). In the following, these variables are called *domain variables*. A finite domain term can then be built using the traditional operations on integers.

Definition 1 An FD-term is defined inductively as follows:

1. a domain variable is an FD-term;
2. a natural number is an FD-term;
3. if t_1 and t_2 are FD-terms, so are

$$t_1 + t_2, t_1 - t_2, t_1 * t_2, t_1 \; div \; t_2, t_1 \; mod \; t_2.$$

The operators have their standard semantics on integers: in particular, *div* denotes the integer division and *mod* is the remainder of the division.

The constraints on finite terms can now be defined in a simple way.

Definition 2 An FD-constraint is a constraint

$\mathtt{x} \in \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}$,
$\mathtt{x} \in \mathtt{min..max}$,
$\mathtt{x} \notin \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}$,
$\mathtt{x} \notin \mathtt{min..max}$,
$\mathtt{t}_1 > \mathtt{t}_2$, $\mathtt{t}_1 \geq \mathtt{t}_2$, $\mathtt{t}_1 = \mathtt{t}_2$, $\mathtt{t}_1 \neq \mathtt{t}_2$, $\mathtt{t}_1 \leq \mathtt{t}_2$, $\mathtt{t}_1 < \mathtt{t}_2$.

where `x` is a domain variable, $\mathtt{v}_1, \ldots, \mathtt{v}_n, \mathtt{min}, \mathtt{max}$ are positive integers, and $\mathtt{t}_1$, $\mathtt{t}_2$ are FD-terms. Once again, the semantics of the relations is the standard one.

[2] Several combinators of `cc(FD)` permit more sophisticated search procedures.

Since deciding FD-constraints is NP-complete, the constraint solver in cc(FD) is incomplete and enforces arc-consistency [16] on the constraints (instead of full consistency). Intuitively, arc-consistency reduces the domains of the variables by removing values which are not locally consistent.

Definition 3 Let $x_1, \ldots, x_n$ be variables with domains $D_1, \ldots, D_n$. A constraint $c(x_1, \ldots, x_n)$ is arc-consistent wrt $D_1, \ldots, D_n$ iff, for each variable x_i and value $v_i \in D_i$, there exist values $v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}, v_{i+1} \in D_{i+1}, \ldots, v_n \in D_n$ such that $c(v_1, \ldots, v_n)$ holds.

Definition 4 Let $x_1, \ldots, x_n$ be domain variables with domains $D_1, \ldots, D_n$. A set S of constraints over $x_1, \ldots, x_n$ is arc-consistent wrt $D_1, \ldots, D_n$ iff each constraint $c \in S$ over $x_{i_1}, \ldots, x_{i_m}$ is arc-consistent wrt $D_{i_1}, \ldots, D_{i_m}$.

Similarly, cc(FD) does not use full entailment but the weaker notion of arc-entailment.

Definition 5 A constraint $c(x_1, \ldots, x_n)$ is arc 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 constraint solver of cc(FD) is based on AC-5 [25], a generic arc-consistency algorithm which can be specialized to AC-4 [18] and AC-3 [16]. In addition, AC-5 can be instantiated to produce an $O(cd)$ algorithm, where c is the number of constraints and d is the size of the largest domain, for many classes of binary constraints, including functional and monotone constraints and their piecewise generalizations.

Example 6 The constraint solver of cc(FD) can be applied naturally to solve PERT problems. Consider the following program:

```
stateConstraint([Sa,Sb,Sc,Sd,Se,Sf,Sg,Sh,Sj,Sk,Send],Send) :-
   stateDomain([Sa,Sb,Sc,Sd,Se,Sf,Sg,Sh,Sj,Sk,Send]),
   Sb ≥ Sa + 7, Sd ≥ Sa + 7, Sc ≥ Sb + 3,
   Se ≥ Sc + 1, Se ≥ Sd + 8, Sg ≥ Sc + 1,
   Sg ≥ Sd + 8, Sf ≥ Sd + 8, Sf ≥ Sc + 1,
   Sh ≥ Sf + 1, Sj ≥ Sh + 3, Sk ≥ Sg + 1,
   Sk ≥ Se + 1, Sk ≥ Sj + 2, Send ≥ Sk + 1.

stateDomain([]).
stateDomain([F|T]) :-
   F ∈ 0..30,
   stateDomain(T).
```

Each variable corresponds to the start date of a task and each of the constraints expresses a precedence constraint between two tasks (e.g. the first constraint expresses that task b must be scheduled after task a whose duration is 7). The predicate stateConstraint reduces, by constraint propagation, the domains of the variables to Sa ∈ 0..8, Sb ∈ 7..19, Sc

$\in$ `10..22`, `Sd` $\in$ `7..15`, `Se` $\in$ `15..27`, `Sf` $\in$ `15..23`, `Sg` $\in$ `15..18`, `Sh` $\in$ `15..27`, `Sj` $\in$ `19..27`, `Sk` $\in$ `21..29`, and `Send` $\in$ `22..30`. In addition, the assignment of the minimum possible value to the end task would automatically solve the PERT problem.

Note however that constraints are not generally stated so explicitly; they are often generated dynamically by a recursive predicate to guarantee the genericity of the program. For instance, precedence constraints can be enforced by a predicate of the form:

```
statePrecedence([]).
statePrecedence([precedence(Start1,Start2,Duration2)|Prec]) :-
   Start1 ≥ Start2 + Duration2,
   statePrecedence(Prec).
```

The predicate `StatePrecedence` receives a list of terms describing the precedence constraints and enforces the associated constraints.

2.4 The Cardinality Combinator

In this section, we discuss the cardinality operator, an important abstraction used in the two applications described later. The syntax of the cardinality operator is

$$\#(l, [c_1, \ldots, c_n], u)$$

where l, u are integers or domain variables and $c_1, \ldots, c_n$ are constraints or cardinality combinators. The declarative semantics of the cardinality operator is as follows: the number of true formula in $[c_1, \ldots, c_n]$ is no less than l and no more than u. Note that the cardinality operator generalizes many other logical connectives as shown by the following equivalences:

$c_1 \wedge \ldots \wedge c_n$ is equivalent to $\#(n, [c_1, \ldots, c_n], n)$;
$c_1 \vee \ldots \vee c_n$ is equivalent to $\#(1, [c_1, \ldots, c_n], n)$;
$\neg c$ is equivalent to $\#(0, [c], 0)$.

Implication and equivalence can now be obtained in a straightforward way. `cc(FD)` allows the programmer to use the standard logical connectives as abbreviations for cardinality formulas.

The main interest of the cardinality combinator lies in its operational semantics. The combinator implements a principle well known in operations research and artificial intelligence: "infer simple constraints from difficult ones." The intuitive idea is to make sure that the cardinality combinator can be satisfied in some way. Moreover, if there is only one way to satisfy it, then the constraints necessary to satisfy it are introduced in the constraint store. Constraint entailment, i.e. checking if a constraint is implied by a set of constraints, is used to check if there is a way to satisfy the constraints.

Example 7 In scheduling applications, disjunctive constraints arise when two tasks cannot be scheduled at the same time because, for instance, they use the same resource. In [23, 10], disjunctive constraints were expressed through a non-deterministic procedure as follows:

```
disjunctive(S1,D1,S2,D2) :-
   S1 ≥ S2 + D2.
disjunctive(S1,D1,S2,D2) :-
   S2 ≥ S1 + D1.
```

where S1,S2 are the starting dates of two tasks and D1,D2 their respective durations. Operationally, executing such a procedure amounts to creating a choice point: the first task is chosen to be scheduled after the second task. If backtracking occurs later on, the second alternative (i.e. the first task is scheduled before the second one) is selected. The cardinality operator offers another possibility for expressing disjunctive constraints:

```
disjunctive(S1,D1,S2,D2) :-
   #(1, [ S1 ≥ S2 + D2, S2 ≥ S1 + D1 ], 2).
```

The cardinality formula expresses the fact that the first task is scheduled after the second task or vice-versa. Operationally, the cardinality formula makes sure that at least one of possibilities is (arc-)consistent with the constraint store. If one possibility is ruled out, the other possibility is automatically enforced. For instance, the predicate show defined as follows:

```
show(S1,S2) :-
   S1 ∈ 1..5,
   S2 ∈ 1..2,
   #(1, [ S2 ≥ S1 + 4 , S1 ≥ S2 + 4 ], 2).
```

automatically assigns S2 to 1 and S1 to 5 without any backtracking.

More sophisticated handlings of disjunctions, taking for instance several tasks simultaneously, are possible in cc(FD) using, for instance, the cardinality combinator in conjunction with the implication combinator [27]. Note also that, in the perfect square problem, the above handling of disjunctions is generalized to 2 dimensions.

Example 8 The cardinality combinator also enables to link Boolean variables (i.e. 0-1 domain variables) with a constraint in the following way:

```
B ∈ 0..1, #( B, [C], B ).
```

The above expression makes sure that

1. the Boolean B is instantiated to 1 (resp. 0) when the constraint C (resp.its negation) is entailed by the constraint store.

2. the constraint C (resp. its negation) is added to the constraint store when B is instantiated to 1 (resp. 0).

This technique is used in the perfect square problem and in the DSP applications to implement various forms of capacity constraints.

2.5 Optimization

cc(FD) contains several meta-level predicates to optimize an objective function. Two common forms are

```
minof(Goal,Function,Res),
maxof(Goal,Function,Res).
```

The predicates expect a goal as first argument, an FD-term as second argument, and a variable as third argument. The predicate `minof` (resp. `maxof`) searches for a solution to `Goal` which minimizes (resp. maximizes) the objective function `function`; the variable `Res` is instantiated to such a solution. Operationally, the implementation uses a depth-first branch and bound algorithm. The idea is to search for a first solution. Each time a solution with cost `C` is obtained, a constraint `Function` $<$ `C` (resp. `Function` $>$ `C`) is generated dynamically. The search is completed when the search space has been explored implicitly.

Example 9 Reconsider for instance the PERT problem discussed previously. A minimal solution instantiating all variables can be obtained by the following program:

```
solvePert(ResStartDates) :-
   stateConstraint(StartDates,EndDate),
   minof(labeling(StartDates),EndDate,ResStartDates).

labeling([]).
labeling([F|T]) :-
   indomain(F),
   labeling(T).
```

The key idea here is to state all constraints in a deterministic way and to enclose the nondeterministic part (i.e. the predicate making choices) in the meta-level predicate for optimization. The `labeling` predicate in the above program simply instantiates all variables using the `indomain` predicate. The `indomain` predicate is a nondeterministic predicate trying all possible values for the variable. Note that, in the above program, `cc(FD)` finds the optimal solution without any backtracking. For disjunctive scheduling, the labeling generally contains also a procedure to choose the ordering between the tasks [10, 23].

3 The Perfect Square Problem

3.1 Problem Statement

The problem amounts to packing a number of squares, all of different sizes, in a larger square, called the master square, in such a way that the squares do not overlap and leave no empty space. Placing 21 squares in a master square is of particular interest since 21 is the smallest number of squares that can fit in a square. This problem and its variations have attracted much attention in the CLP community. Colmerauer [6] presents a program to fill a rectangle with squares of different sizes using linear equations, inequalities, and disequations over rational numbers. Aggoun and Beldiceanu [1] present a finite domain program which can solve large instances (e.g. 21 and 24) of the perfect square problem when the sizes of the squares are given. The program uses a specialized cumulative constraint and exploits the link between cumulative constraints and packing in two dimensions. We propose a simple `cc(FD)` program implementing a natural problem formulation which can solve large instances (e.g. 21 and 24) in about 30 seconds on a SUN 4/60.

3.2 Problem Solution

The main idea behind the program consists in creating the squares (i.e. creating variables for their coordinates), expressing the non-overlapping constraints, and assigning values for the coordinates in a nondeterministic way. Moreover, redundant capacity constraints are generated to speed up the computation by pruning the search space early.

3.2.1 Problem Data

The program is generic and receives as data the size of the master square and the sizes of the squares. For instance the data for 21 squares is as follows:

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

3.2.2 Program Variables

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 1 and $S - S_i + 1$ 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) :-
   Sdate := Size - S + 1,
   X ∈ 1..Sdate,
   Y ∈ 1..Sdate,
   generateCoordinates(Xs,Ys,Ss,Size).
```

3.2.3 The Non-Overlapping Constraint

The non-overlapping 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 durations can be expressed using the cardinality constraint:

```
nooverlap(X1,Y1,S1,X2,Y2,S2) :-
   #(1,[X1 + S1 ≤ X2, X2 + S2 ≤ X1, Y1 + S1 ≤ Y2, Y2 + S2 ≤ Y1],2).
```

The cardinality constraint simply expresses that the first square is on the left, on the right, below, or above the second square. It makes sure that, as soon as three possibilities are ruled out (i.e. they are not consistent with the constraint store), the last option is automatically enforced. As mentioned previously, the definition simply generalizes the traditional disjunctive constraints to two dimensions.

3.2.4 The Capacity Constraints

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 domain 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 constraint:

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

The surrogate constraint 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).
```

3.2.5 The Labeling Procedure

The generation of positions for the squares requires to give values to the coordinates of all squares. We use the idea of [1] for the labeling, 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([Sq|Sqs]) :-
   minlist([Sq|Sqs],Min),
   selectSquare([Sq|Sqs],Min,Rest),
   labeling(Rest).

selectSquare([Sq|Sqs],Min,Sqs) :-
   Sq = Min.
selectSquare([Sq|Sqs],Min,[Sq|Rest]) :-
```

```
Sq > Min,
selectSquare(Sqs,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.

3.2.6 The Overall Program

Figure 3 shows the overall program for the perfect square problem. As should be clear, the program is rather small. It packs 21 or 24 squares in a master square in about 30 seconds on a SUN 4/60, illustrating the expressiveness and efficiency of the language.

4 The Digital Signal Processing Application

4.1 Problem Statement

The availability of programmable digital processors at low cost has raised much interest in DSP and its applications such as speech, image processing, and noise and echo cancellation. Moreover, the design of multiprocessor DSP systems has emerged as one of the most promising directions. Recently, Chinneck and al. [4, 15] have proposed a new approach to the design of DSP applications, that couples an extensible architecture with an automatic task to processor scheduling method. This approach is suitable for many complex DSP applications as it combines some of the advantages of specialized systems (often efficient but not flexible) with those of general-purpose systems (often flexible but much less efficient). In addition, it provides a real-time simulation tool for systems that will eventually be implemented on a VLSI chip. In the rest of this section, we review the key ideas behind the architecture and introduce the main combinatorial search problems. The presentation is based on [4, 15] where complete information on the approach can be found.

4.1.1 The Architecture

The extensible architecture is depicted in Figure 4 from [15]. It captures two common paradigms of DSP applications: pipelined processing and master-slave processing. The master (processor 1) has a direct communication link with all processors in the pipeline whereas processor i in the pipeline can read from its predecessor (processor $i-1$) and write to its successor (processor $i+1$). Some redundant links are included in this architecture to simplify programming of the applications.

The processors are synchronized by a clock corresponding to the sampling rate of the incoming signal (or data) of the system. The clock cycle is divided into a *read* part during which a processor copies its input buffers into local memory and a *write* part during which the processor is allowed to write in its output buffers. As a consequence, all tasks must be performed during the clock cycle and data to be transferred must be available at the clock edge.

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

generateSquares(Xs,Ys,Sizes,Size) :-
   sizeMaster(Size), sizeSquares(Sizes),
   generateCoordinates(Xs,Ys,Sizes,Size).
generateCoordinates([],[],[],_).
generateCoordinates([X|Xs],[Y|Ys],[S|Ss],Size) :-
   Sdate := Size - S + 1,
   X ∈ 1..Sdate, Y ∈ 1..Sdate,
   generateCoordinates(Xs,Ys,Ss,Size).

stateNoOverlap([],[],nnnnnnnnnn[]).
stateNoOverlap([X|Xs],[Y|Ys],[S|Ss]) :-
   stateNoOverlap(X,Y,S,Xs,Ys,Ss),
   stateNoOverlap(Xs,Ys,Ss).
stateNoOverlap(X,Y,S,[],[],[]).
stateNoOverlap(X,Y,S,[X1|Xs],[Y1|Ys],[S|Ss]) :-
   nooverlap(X,Y,S,X1,Y1,S1),
   stateNoOverlap(X,Y,S,Xs,Ys,Ss).
nooverlap(X1,Y1,S1,X2,Y2,S2) :- #(1,[X1 + S1 ≤ X2, X2 + S2 ≤ X1, Y1 + S1 ≤ Y2, Y2 + S2 ≤ Y1],2).

stateCapacity(Cs,Sizes,Size) :- stateCapacity(1,Size,Cs,Sizes).
stateCapacity(Pos,Size,Cs,Sizes) :- Pos > Size.
stateCapacity(Pos,Size,Cs,Sizes) :-
   capacity(Pos,Cs,Sizes,Size),
   Pos1 := Pos + 1,
   stateCapacity(Pos1,Size,Cs,Sizes).
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).

labeling([]).
labeling([Sq|Sqs]) :-
   minlist([Sq|Sqs],Min),
   selectSquare([Sq|Sqs],Min,Rest),
   labeling(Rest).
selectSquare([Sq|Sqs],Min,Ss) :- Sq = Min.
selectSquare([Sq|Sqs],Min,[Sq|Rest]) :-
   Sq > Min,
   selectSquare(Sqs,Min,Rest).
```

Figure 3: The Perfect Square Program

Figure 4: The Extensible Architecture

Each processor in the architecture is identical, has its own local memory, and is characterized by its limits on the data memory, the code memory, and the processing time in a synchronization cycle.

The main features of the architecture are

1. *Extensibility*: more processors can be added to accommodate the need for more processing power.

2. *Bounded Communication Delay:* the communication delay between two processors is at most 2.

In the following, we assume that the DSP application is deterministic wrt the data stream and the task-to-processor scheduling can be performed once in the life time of the system for fixed systems or once for each configuration for reconfigurable systems.

4.1.2 The Task Graph

The DSP application is described by a task graph where each node represents a DSP function and an arc denotes a data flow path between two functions. Each task is characterized by its demand in processing time (e.g. 80 $\mu s.$), in data memory (e.g. 50 bytes) and in code memory (e.g. 10 bytes). The data flow path expresses precedence constraints between the tasks.

The communication delay between two successive tasks depends on which processors the tasks are assigned to. If the two tasks are assigned to the same processor, the communication delay is zero. If the two tasks are assigned to adjacent processors (i.e. i for the origin task and $i+1$ for the destination task), the communication delay is one clock cycle, i.e. the destination task can proceed at the next clock cycle. Otherwise the communication is two clock cycles since the data transfer must go through the master.

In the following, we assume that the whole task graph include two fictitious tasks: an input task and an output task. Both tasks are assigned to the master processor and have no requirement. The total delay of the application is the time in clock cycles between the input and output tasks.

4.1.3 The Combinatorial Search Problems

There are two orthogonal problems that need to be solved:

1. minimizing the number of processors for a task graph;
2. minimizing the total delay for a given number of processor.

The first problem is a generalization of the well known bin-packing problem for which there exist very good approximation algorithms (e.g. first-fit decreasing). Our `cc(FD)` program uses such an algorithm to obtain a solution to the first problem. The second problem (i.e. finding the minimum total delay given the number of processors given by the first problem) is more difficult and is the topic of the remaining part of this section.

4.2 Problem Solution

4.2.1 Overview of the Program

The program is once again a typical finite domain program. The key idea is to create the problem variables (i.e. those variables on which the constraints will be imposed), to state the constraints and to generate values for the variables with a labeling procedure. Since it is an optimization problem, the labeling procedure, together with an objective function, is enclosed inside a meta-predicate for finding the optimal solution. The main predicate of the program is as follows:

```
scheduleDsp(ResTasks) :-
   createTasks(Tasks,Nbprocs),

   statePrecedence(Tasks),
   stateCapacity(Tasks,Nbprocs),
   stateDelay(Tasks),
   stateSurrogate(Tasks),

   findEndTask(Tasks,EndTask),
   getCycle(EndDate,EndCycle),

   minof(labeling(Tasks),EndCycle,ResTasks).
```

The first goal creates the tasks `Tasks`, the next four goals state the constraints including some "semantically" redundant constraints, the next two goals retrieve the variable expressing the total delay (i.e. the cycle of the end task), and the last goal is the meta-predicate for optimization, receiving the labeling procedure and the variable `EndCycle` as objective function. In the rest of this section, the above steps are explained in details.

4.2.2 Problem Variables

Each task in the DSP application is associated with two variables:

- P_i: the processor to which task i is assigned;
- S_i: the starting cycle of task i.

In the following, we refer to the first type of variables as `processor` variables and the second type of variables as `cycle` variables.

The key problem to be solved is the assignment of processors to the tasks. Once this assignment is completed, it is a simple matter to find a solution to the overall problem. Indeed, the problem is then reduced to a simple PERT problem which is solved automatically through constraint propagation as mentioned previously. A solution can then be obtaining by assigning the earliest starting cycle to each task.

Our `cc(FD)` program creates a term `task(Number,Name,P,S,Time,Memory,Code)` for each task where `number` is the number of the task, `name` its name, `P` its processor, `S` its starting cycle, and `Time, Memory, Code` its demands in computing time, data memory, and code memory. All components of the term are ground (i.e. fully instantiated) except `P` which is constrained to range between 1 and the number of processors available and `S` which is required to take a value between 0 and the maximum delay (which can be computed easily by assuming the worst communication delays).

4.3 Precedence Constraints

As mentioned previously, the task graph implies a number of precedence relationships between the tasks. The precedence constraints only affect the cycle variables and are stated in the standard way.

```
precedence(Task1,Task2) :-
   getCycle(Task1,C1),
   getCycle(Task2,C2),
   C1 ≤ C2.
```

4.4 Capacity Constraints

Since the capacity of each processor is limited, it is necessary to make sure that the tasks assigned to a processor do not exceed its resources. The capacity constraints for the DSP application are closely related to the capacity constraints of the perfect square problem but uses inequalities instead of equations. The `cc(FD)` program associates a Boolean variable with each task and each processor. The Boolean variable is true iff the task is assigned to the processor and is obtained through a cardinality formula

$$\#(\mathtt{B}_i,[\ \mathtt{P}_i\ =\ \mathtt{P}\ ],\ \mathtt{B}_i)$$

where $\mathtt{B}_i$ is the Boolean variable associated with task i, P_i is the processor variable of task i, and `P` is a processor number. The inequalities can now be stated as follows:

```
Ti₁ * B₁ + ... + Tiₙ * Bₙ ≤ MaxTime,
Me₁ * B₁ + ... + Meₙ * Bₙ ≤ MaxMem,
Ti₁ * B₁ + ... + Tiₙ * Bₙ ≤ MaxCode,
```

where $B_1,\ldots,B_n$ are the Boolean variables associated with all tasks for processor P and the Ti_k, Me_k, Ce_k are the demands of each task k in time, data memory, and code memory.

The program for generating a capacity constraint is as follows:

```
capacity(Tasks,Proc,MaxTime,MaxMemory,MaxCode) :-
   collectCapacity(Tasks,Value,Time,Memory,Code),
   Time ≤ MaxTime,
   Memory ≤ MaxMemory,
   Code ≤ MaxCode.

collectCapacity([],Proc,0,0,0).
collectCapacity([Task|Tasks],Proc,T * B + Ts, M * B + Ms, C * B + Cs) :-
   getTime(Task,T),
   getMemory(Task,M),
   getCode(Task,C),
   getProc(Task,P),
   B ∈ 0..1,
   #( B, [ P = Proc ], B ),
   collectCapacity(Tasks,Proc,Ts,Ms,Cs).
```

The predicate collectCapacity collects linear terms for the data memory, the code memory, and processing time. Those terms are then used in predicate capacity to state the capacity constraints. Collecting the linear terms is performed in a way similar to the capacity constraints for the perfect square problem.

4.4.1 The Delay Constraints

The total communication delay depends on the delay between every two successive tasks in the task graph. The cc(FD) program generates delay constraints between the cycle variables of the tasks. If C_1, C_2 are cycle variables of two successive tasks, the idea is to generate a constraint

$$C_2 \geq C_1 + \text{Delay}$$

where Delay is the communication delay between the two tasks. The communication delay can be expressed as a constraint involving the processor variables P_1, P_2 of the tasks using cardinality formulas:

```
delay(P1,P2,S1,S2) :-
   Delay ∈ 0..2,
   Delay = 0 ⇔ P1 = P2,
   Delay = 1 ⇔ P1 ≠ P2 & (P1 = 1 ∨ P2 = 1 ∨ P2 = P1 + 1),
   C2 ≥ C1 + Delay.
```

The first goal expresses that the delay is atmost 2, the second goal states that the delay is zero whenever the tasks are allocated to the same processor, while the third goal imposes the delay to be one whenever the tasks are not allocated to the same processor and either one of them is assigned to the master processor or the second task is allocated to a processor on the right of the processor of the first task. The third case can be omitted since the first two cases are expressed as equivalences and the delay is at most 2. Note that the distance constraints preclude the need for the precedence constraints which can now be omitted. Finally note the simplicity of the implementation which follows closely the definition.

4.4.2 Surrogate Constraints

The cycle variable of the end task provides an opportunistic evaluation of the minimum delay of the DSP problem. Indeed, this variable is constrained by all the delay constraints and its minimum value provides the optimistic evaluation. It is however possible and desirable to improve this evaluation by stating some properties of the solutions. This helps pruning the search space during the search for the optimal solution. The key idea is to exploit the resource constraints jointly with the precedence constraints. Consider, for instance, a precedence path $t_1, t_2, \ldots, t_n$ and a resource r (e.g. computing time, data memory, code memory). If the tasks $t_1, \ldots, t_i$ $(1 \leq i \leq n)$ have resource requirements for resource r exceeding the capacity of the processor, it follows that the delay between tasks t_1 and t_i is at least 1. A constraint

$$\mathtt{C}_i > \mathtt{C}_1$$

where $\mathtt{C}_i$,$\mathtt{C}_1$ are the cycle variables of tasks t_i and t_1 can then be generated without removing any of the solutions.

Our `cc(FD)` program systematically generates all such surrogate constraints for all resources. Note that it is sufficient to consider for each task the smallest subpath exceeding the capacity of the resources.

4.4.3 The Labeling Procedure

We now consider the labeling procedure for the DSP application. As mentioned previously, the key issue is the assignment of a processor to each task. It is a simple matter to complete the solution by assigning the earliest values to the cycle variables. Hence we focus on the first part of the labeling.

As is typical with constraint satisfaction problems, two decisions need to be made:

1. which variable should be instantiated first;

2. which value should be given to the selected variable.

For the first decision, two strategies have been included in the `cc(FD)` program:

- a depth-first labeling;

- a breath-first labeling.

The depth-first labeling proceeds by assigning processor variables in a depth-first traversal of the task graph starting with the start task while the breath-first labeling uses a breath-first traversal. The motivation behind the depth-first strategy is to obtain quickly an accurate evaluation of the total delay. The motivation behind the breath-first strategy is to exploit the locality of the tasks and to schedule closely related tasks together. Both strategies have been systematically evaluated and experimental results are discussed in the next section.

As far as the second decision is concerned, the `cc(FD)` program uses the following heuristics: if p is the value of the previously assigned variable,

- first try p;
- next try $p + 1$;
- next try the master processor;
- finally try all other values.

The heuristics is motivated by the desire to minimize the total communication delay. The first choice guarantees a communication delay of zero between the last two tasks, the second and third choices ensure a communication delay of 1 while the last choice imposes a communication delay of 2. The implementation of this heuristics is based on the following procedure:

```
assign(P,Previous) :-
   P = Previous.
assign(P,Previous) :-
   P = Previous + 1.
assign(P,Previous) :-
   P ≠ Previous,
   P = 1.
assign(P,Previous) :-
   P ≠ Previous,
   P ≠ Previous + 1,
   P ≠ 1,
   indomain(P).
```

4.4.4 Exploiting Symmetries

In general, DSP applications have many symmetrical solutions and it would be of great advantage if the program could exploit them in order to prune the search space. Consider for instance the computation step where the program tries to assign a processor to a task. If there are several processors unused at this computation step, it would be convenient to restrict attention to only one of them. It turns out[4] that this idea is only applicable under restricted circumstances, namely when all tasks which are on a parallel pipeline with the task being considered have been assigned already.

To evaluate the possible gain of symmetries in the DSP applications, our program also includes an *incomplete* labeling procedure that systematically performs the above pruning at the risk of missing the optimal solution. The labeling procedure needs to keep track of the used processors and is based on the following procedure:

```
assign(P,Previous,MaxUsed,MaxUsed) :-
   P = Previous.
assign(P,Previous,MaxUsed,P) :-
   P = MaxUsed + 1.
assign(P,Previous,MaxUsed,MaxUsed) :-
   P ≠ Previous,
   P = 1.
assign(P,Previous,MaxUsed,MaxUsed) :-
   P ≠ Previous,
   P ≠ 1,
   P ≤ MaxUsed,
   indomain(P).
```

In this procedure, the third argument is the maximum number of the used processors. The fourth argument is the new maximum number. The last clause only considers the previously used processors. Hence only one new processor can be used (i.e. in the second clauses).

4.5 Experimental Results

In this section, we summarize the experimental results obtained by the cc(FD) program on actual DSP applications and compare them with a specialized branch and bound algorithm (written in C).

First note that the overall cc(FD) program is about 4 pages long. This obviously comes from the fact that the search tree management and the constraint propagation are automatically taken care by the programming language. The programming task consisted of stating the constraints and coming with a suitable labeling procedure. Note also the adequacy of the cardinality operator to express sophisticated constraints such as the capacity and delay constraints.

Table 1 presents a description of actual DSP applications taken from [15]. The first column describes the name of the applications, the second column the size of the task graph, the third column the number of processors available and the last column the topology of the applications. The first problems are small, the last one being already much larger. For instance, the potential search space of RDAD40 is 8^{25} which is 2^{75}.

Table 2 presents the results of the complete cc(FD) algorithm. The second column gives the total delay of the optimal solution, the third column the times of the cc(FD) program with the depth-first labeling strategy, the fourth column the times of the cc(FD) program with the breath-first labeling strategy, and the fifth column the times of the specialized branch and bound of [15]. All times are given in seconds for a SUN 4/60. On these actual applications, the cc(FD) program behaves much better in general with the depth-first strategy than with the breath-first strategy. In particular, it is about 14, 2230, 59 times faster on RDAD04, RDAD08, and RDAD40. The cc(FD) program also compares well with the specific branch

Problem	Size	Processors	Topology
RDAD01	9	3	Pipeline
RDAD02	9	3	Pipeline
RDAD03	6	5	Architecture-like
RDAD04	19	6	Parallel Pipelines
RDAD05	12	4	Pipeline
RDAD06	16	5	Parallel Pipelines
RDAD07	12	4	Parallel Pipelines
RDAD08	15	5	Merging Tasks
RDAD09	9	6	Many Generators
RDAD10	15	5	Parallel Pipelines
RDAD20	13	5	Architecture-like
RDAD40	25	8	Parallel Pipelines
RDAD41	25	8	Parallel Pipelines

Table 1: Description of Actual DSP Applications

Problem	Total Delay	`cc(FD)`: DF	`cc(FD)`: BF	Specialized BB
RDAD01	3	0.78	0.31	0.016
RDAD02	3	0.72	0.32	0.016
RDAD03	5	0.22	0.11	0.000
RDAD04	3	1.66	24.14	51.700
RDAD05	3	1.12	5.29	0.016
RDAD06	3	2.68	18.31	6.300
RDAD07	2	0.56	5.19	0.050
RDAD08	3	3.23	7202.00	963.13
RDAD09	2	0.18	0.20	0.016
RDAD10	4	3.90	1.80	0.033
RDAD20	3	0.99	0.78	0.016
RDAD40	5	54.40	3216.7	?????
RDAD41	4	4.24	3.16	0.100

Table 2: Results on Actual DSP Applications

Problem	Delay-DF	Time-DF	Delay-BF	Time-BF
RDAD01	3	0.78	3	0.26
RDAD02	3	0.74	3	0.27
RDAD03	5	0.18	5	0.11
RDAD04	3	1.25	5	4.61
RDAD05	3	0.92	5	1.62
RDAD06	3	2.12	3	5.88
RDAD07	2	0.56	2	1.83
RDAD08	3	1.86	3	359.69
RDAD09	2	0.16	3	0.15
RDAD10	4	3.03	4	1.13
RDAD20	3	0.79	3	0.52
RDAD40	5	16.55	5	150.16
RDAD41	4	3.45	4	1.46

Table 3: Results With Symmetries on Actual DSP Applications

and bound algorithm. In particular, the DF program is 31 and 298 faster on RDAD04 and RDAD08. The specific branch and bound algorithm also could not find a solution for RDAD40 but is faster in general on the problems requiring less computing time.

Table 3 presents the results of the incomplete `cc(FD)` algorithm, i.e. the algorithm exploiting systematically symmetries. `Delay-DF` and `Time-DF` give the best total delay found and the computation time for the `cc(FD)` program with the depth-first search strategy while `Delay-BF` and `Time-BF` the total delay and the computation time for the `cc(FD)` program with the breath-first search strategy. The first point to notice that, on those problems, the optimal solution is always found by both programs. The computation time can also be drastically reduced as illustrated by the problems `RDAD40` and `RDAD08`. To compare the results with the specialized branch and bound, we give in Table 4 the results with the same exploitation of the heuristics and, in addition, a dynamic clustering of tasks. The dynamic clustering groups together tasks on a pipeline and thus substantially reduces the search space. The results indicate that the `cc(FD)` programs, even without dynamic task clustering, still performs well. It is faster on `RDAD08` and less than 10 times slower on `RDAD40` where many task clusterings can take place. These results indicate clearly that a language like `cc(FD)` can be competitive for industrial applications while providing a short development time.

5 Conclusion

In this paper, we have described the application of `cc(FD)` to two applications: the perfect square problem and a digital signal processing application. `cc(FD)` is a declarative constraint logic programming language over finite domains offering a short development time and a competitive efficiency for many discrete combinatorial search problems. The `cc(FD)` program for the perfect square problem is about a page long, conceptually simple, and packs 21 or 24 squares in about 30 seconds on a SUN 4/60. The `cc(FD)` program for the DSP applications is about 4 pages long and is competitive in efficiency with a specialized branch

Problems	Delay-SBB	Time-SBB
RDAD01	3	0.016
RDAD02	3	0.000
RDAD03	5	0.000
RDAD04	3	0.000
RDAD05	3	0.016
RDAD06	3	0.033
RDAD07	2	0.000
RDAD08	3	29.067
RDAD09	2	0.000
RDAD10	5	0.000
RDAD20	3	0.000
RDAD40	5	1.733
RDAD41	4	0.033

Table 4: Results of the Specialized Branch and Bound with Symmetries and Clustering

and bound algorithms developed for the task. Both programs make use of advanced features of `cc(FD)` (e.g. the cardinality operator) and exploit the integration of constraint solving in a declarative language, e.g. by using sophisticated choice procedures and generating surrogate constraints. Both applications illustrate the potential benefit of constraint languages such as `cc(FD)`.

Acknowledgements

Michel Van Caneghem and Alain Colmerauer suggested the perfect square application and the approach taken in the `cc(FD)` program and pointed out the reference [1]. Gerald Karam provided the DSP applications and gave us the benchmarks to test our programs. We would like to thank them as well as Yves Deville and Vijay Saraswat for the work on the design of `cc(FD)`. 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] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

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

[4] 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.

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

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

[7] 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.

[8] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[9] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[10] 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.

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

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

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

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

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] P. Van Hentenryck. A Logic Language for Combinatorial Optimization. *Annals of Operations Research*, 21:247–274, 1989.

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

[24] 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.

[25] P. Van Hentenryck, Y. Deville, and C.M. Teng. A Generic Arc Consistency Algorithm and its Specializations. *Artificial Intelligence*, 1992. Accepted for publication.

[26] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 1992. To Appear. A preliminary version was presented at The International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Fl, January 1990.

[27] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence*, 1992. (Special Issue on Constraint-Based Reasoning), accepted for publication.