

Search and Strategies in OPL

PASCAL VAN HENTENRYCK

Brown University

LAURENT PERRON AND JEAN-FRANCOIS PUGET

Ilog SA

OPL is a modeling language for mathematical programming and combinatorial optimization. It is the first language to combine high-level algebraic and set notations from mathematical modeling languages with a rich constraint language and the ability to specify search procedures and strategies that are the essence of constraint programming. This paper describes the facilities available in OPL to specify search procedures. It describes the abstractions of OPL to specify both the search tree (search) and how to explore it (strategies). The paper also illustrates how to use these high-level constructs to implement traditional search procedures in constraint programming and scheduling.

Categories and Subject Descriptors: D.3 [Software]: Programming Languages; D.3.2 [Programming Languages]: Languages Classifications—*Constraint and logic languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Constraints; control structures*

General Terms: Design, Languages

Additional Key Words and Phrases: Constraint programming, modeling languages, search, combinatorial optimization

1. INTRODUCTION

Combinatorial optimization problems are ubiquitous in many practical applications, including scheduling, resource allocation, planning, and configuration problems. These problems are computationally difficult (i.e., they are NP-hard) and require considerable expertise in optimization, software engineering, and the application domain.

The last two decades have witnessed substantial development in tools to simplify the design and implementation of combinatorial optimization problems. Their goal is to decrease development time substantially while preserving most of the efficiency of specialized programs. Most tools can be classified in two categories: mathematical modeling languages and constraint programming languages. Modeling languages such as AMPL [Fourer et al. 1993] and GAMS [Bisschop and Meeraus 1982] provide high-level algebraic and set notations to express, in concise ways, mathematical problems that can then be solved

Pascal Van Hentenryck was supported in part by an NSF National Young Investigator Award.

Authors' addresses: P. Van Hentenryck, Brown University, Department of Computer Science, Box 1910, Providence, RI 02912; Laurent Perron and Jean-François Puget, Ilog SA, 9 rue de Verdun, F-94253 Gentilly, France.

Permission to make digital/hard copies of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2000 ACM 1529-3785/00/1000-0285 \$5.00

using state-of-the-art solvers. These modeling languages do not require specific programming skills and can be used by a wide audience. Constraint programming languages such as CHIP [Dincbas et al. 1988], PROLOG III [Colmerauer 1990], Prolog IV [Colmerauer 1996], Oz [Schulte 1997; Smolka 1995], and ILOG SOLVER [Ilog Solver 4.4 1998] have orthogonal strengths. Their constraint vocabulary and their underlying solvers go beyond traditional linear and nonlinear constraints and support logical, higher-order, and global constraints. They also make it possible to program search procedures to specify how to explore the search space. However, these languages are mostly aimed at computer scientists and often have weaker abstractions for algebraic and set manipulation.

The optimization programming language OPL [Van Hentenryck 1999] is a recent addition in this area, and it originated as an attempt to unify modeling and constraint programming languages, both at the language and at the solver technology level. OPL shares with modeling languages their high-level algebraic and set notations. It also contains some novel functionalities to exploit sparsity in large-scale applications, such as the ability to index arrays with arbitrary data structures. OPL shares with constraint programming languages their rich constraint vocabulary, their support for scheduling and resource allocation problems, and the ability to specify search procedures and strategies. OPL also makes it easy to combine different solver technologies for the same application.

As mentioned, one of the original features of constraint programming is the ability to program search procedures. This ability was present in constraint logic programming since its inception and was directly inherited from logic programming (e.g., [Colmerauer 1990; Jaffar et al. 1992; Van Hentenryck 1989]). It was supported in constraint programming libraries such as ILOG SOLVER [Ilog Solver 4.4 1998]. It has been the subject of much recent research in constraint programming (e.g., [Apt et al. 1998; Laburthe and Caseau 1998; Schulte 1997; Van Hentenryck 1999]), and it is generally considered as one of the most important aspects of constraint programming. In contrast, search procedures are generally absent from modeling languages where only limited support is available to tailor predefined search procedures (e.g., by specifying a static variable ordering). One of the main contributions of OPL is to show how search procedures can be specified, in concise and elegant ways, in modeling languages by building on the strengths of both technologies.

The purpose of this paper is to give an overview of the search abstractions of OPL and to illustrate how they can be used to implement typical search procedures. A search procedure typically consists of two parts: a *search* component defining the search tree to explore and a *strategy* component specifying how to explore this tree.¹ OPL provides high-level abstractions for both components. Its search constructs have many commonalities with recent proposals such as ALMA-0 [Apt et al. 1998] and SALSA [Laburthe and Caseau 1998] that were developed independently, and its strategy constructs, based on the model proposed in [Perron 1999], offer a high-level alternative to the computation states of Oz [Schulte 1997].

¹This paper does not discuss local search. Note however that SALSA [Laburthe and Caseau 1998] has shown that similar abstractions can be used for local search as well.

This paper is about language design: its main insight is the observation that a small number of well-chosen abstractions, in conjunction with the high-level modeling functionalities of OPL, makes it possible to write complex search procedures in concise and natural ways. This claim is substantiated by showing OPL search procedures for representative applications. These search procedures could be written using, say, ILOG SOLVER [Ilog Solver 4.4 1998] (on top of which OPL is implemented), but they would require more development effort and object-oriented programming expertise. As a consequence, it is our belief that OPL contributes to advancing the state-of-the-art in programming search procedures, both in constraint programming and in modeling languages.

The rest of the paper is organized as follows. Section 2 introduces our running examples and serves as a brief introduction to OPL. Section 3 describes the subset of OPL expressions and constraints used to present the search concepts. These two sections aim at making the paper as self-contained as possible. Interested readers may find a comprehensive description of OPL in [Ilog OPL Studio 3.0 2000; Van Hentenryck 1999]. The next two sections are the core of the paper. Section 4 presents the search abstractions of OPL, and Section 5 presents its support for strategies. Section 6 aims at showing the generality of the proposal and illustrates how to use these abstractions to implement typical search procedures in the scheduling domain. Its main objective is to show how the constructs presented in the previous sections apply naturally to scheduling problems, and concepts are only introduced when needed. Section 7 describes related work, and Section 8 concludes the paper. Note that this paper is not intended to be comprehensive (see [Ilog OPL Studio 3.0 2000] for a comprehensive description of the latest version of OPL), but rather it aims at presenting some of the most interesting aspects of search procedures in OPL.

2. THE RUNNING EXAMPLES

This section presents three OPL programs used as running examples for the next two sections. Readers interested in a comprehensive overview of OPL can consult [Van Hentenryck 1999; Van Hentenryck et al. 1999; Van Hentenryck et al. 1999]. The first two programs deal with the n -queens problem, i.e., how to place n queens on a chessboard of size $n \times n$ so that they do not attack each other. This is of course not a real application, but it is appropriate to illustrate, in simple and intuitive ways, many search procedures. Figure 1 describes a simple program based on the observation that there must be a queen on each column of the chessboard. The program thus associates a variable with each column, and its value represents the row on which the queen is placed. The OPL program essentially consists of three parts. The data declaration part specifies the size n of the chessboard (whose value is given in a data file) and declares the range `Domain`. The variable part declares the variables. The constraint part simply states that no two queens can attack each other. In this last part, the keyword `ordered` makes sure that $i < j$ in the `forall` statement. Note that the program does not specify a search procedure. When no search procedure is given, OPL applies a predefined search procedure.²

²The predefined search procedure on this example generates values for the variables using the first-fail principle.

```

int n = ...;
range Domain 1..n;
var Domain queen[Domain];
solve {
  forall(ordered i,j in Domain) {
    queen[i] <> queen[j];
    queen[i] + i <> queen[j] + j;
    queen[i] - i <> queen[j] - j
  };
};

```

Fig. 1. The basic n -queens program.

```

int n = ...;
range Domain 1..n;
enum Dim { col, row };
var Domain queen[Dim,Domain];
solve {
  forall(d in Dim & ordered i,j in Domain) {
    queen[d,i] <> queen[d,j];
    queen[d,i] + i <> queen[d,j] + j;
    queen[d,i] - i <> queen[d,j] - j
  };
  forall(i,v in Domain)
    queen[col,i] = v <=> queen[row,v] = i;
};

```

Fig. 2. Redundant modeling for the n -queens program.

The above program solves large instances of the problem quickly, but it may exhibit wide variations in computation time for two successive values of n . Figure 2 is a first step toward a more stable program. It uses a redundant modeling [Cheng et al. 1996] and models the problem using both a column and a row viewpoint, using constraints to link them together. More specifically, the program uses a two-dimensional array of variables: `queen[col,c]` represents the row of the queen in column c , while `queen[row,r]` represents the column of the queen in row r . The data part specifies an enumerated type `{col,row}` to denote the two viewpoints. The variable part declares a two-dimensional array, whose first dimension is the enumerated type, and the second dimension is the size of the chessboard. The constraint part first states the constraints for both viewpoints and then specifies how to link the two formulations. Once again, there is no search specification in this program.

Our last running example is a warehouse location problem where a company is considering a number of locations for building warehouses to supply its existing stores. Each possible warehouse has a fixed maintenance cost and a maximum capacity specifying how many stores it can support. In addition, each store can be supplied by only one warehouse, and the supply cost to the store varies according to the selected warehouse. The application consists of choosing

```

int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores 0..nbStores-1;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores & w in Warehouses) supplyCost[s,w];

var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;
var int totalCost in 0..maxint;

minimize
  totalCost
subject to {
  totalCost = sum(s in Stores) cost[s] + sum(w in Warehouses)
    fixed * open[w];
  forall(s in Stores)
    cost[s] = supplyCost[s,supplier[s]];
  forall(s in Stores)
    open[supplier[s]] = 1;
  forall(w in Warehouses)
    sum(s in Stores) (supplier[s] = w) <= capacity[w];
};

```

Fig. 3. The warehouse-location program.

which warehouses to build and which of them should supply the various stores in order to minimize the total cost, i.e., the sum of the fixed and supply costs. Figure 3 depicts a program for solving this problem. In this program, the data part specifies the fixed cost `fixed`, the number of stores, and the set of warehouses as an enumerated type. It declares an array specifying the capacity of each warehouse and a two-dimensional array to specify the supply cost from a warehouse to a store, and it computes the maximal supply cost. The variable part declares variables of the form `supplier[s]` to denote the warehouse supplying store `s` and variables of the form `open[w]` to specify whether warehouse `w` is open. It also declares variables to model the supply cost for each store and the total cost. The program of course minimizes the total cost subject to the problem constraints. The first two sets of constraints are used to express the total cost and the transportation cost. The third set of constraints specifies that warehouses supplying stores must be open. The last set of constraints specifies the capacity constraints. As can be seen from these constraints, OPL offers aggregate operators such as `sum` and `prod` as in

$$\text{sum}(i \text{ in } 1..10) a[i] \leq 20$$

which is an abbreviations of

$$a[1] + \dots + a[10] \leq 20$$

OPL also supports higher-order (or meta) constraints (e.g., [Ilog Solver 4.4 1998; Colmerauer 1996; Van Hentenryck and Deville 1991]), i.e., the ability to use constraints inside expressions. For instance, the constraint

```
sum(i in 1..10) (a[i]=3) >= 1
```

ensures that at least an element in $a[1], \dots, a[10]$ is equal to 3. Both aggregate operators and higher-order constraints are exemplified in the capacity constraints of the warehouse location problem. The constraint

```
sum(s in Stores) (supplier[s] = w) <= capacity[w];
```

specifies that the number of stores whose supplier is warehouse w cannot exceed the capacity of w . Note finally that the first two sets of constraints could in fact be avoided by using a more complex objective function. The reason to introduce them is to facilitate the specification of the search procedures later in this paper.

3. OVERVIEW OF CONSTRAINTS AND EXPRESSIONS

This section reviews the subset of OPL expressions and constraints used in the first part of the paper. Since constraints and expressions are orthogonal to the search functionalities, we only consider a small subset of expressions and constraints. Interested readers may consult [Ilog OPL Studio 3.0 2000; Van Hentenryck 1999] for more information. Figure 4 describes the syntax of OPL expressions and constraints used in this paper. The following conventions are used to describe the grammar: terminal symbols are denoted in typewriter font (e.g., `abs`). $\langle nt \rangle$ denotes a nonterminal symbol nt . $[\textit{object}]$ denotes an optional grammar segment \textit{object} . $\{ \textit{object} \}$ denotes zero, one, or several times the grammar segment \textit{object} . \textit{object}^+ denotes an expression

$$\textit{object} \{ \textit{ , } \textit{object} \}$$

while \textit{object}^* denotes an expression $\textit{object} \{ \textit{ ; } \textit{object} \}$.

Expressions in this subset of OPL are constructed from integers, identifiers, possibly indexed (arrays) or dereferenced (records and method invocations), the traditional arithmetic operators, the absolute value operator `abs`, the aggregate operator `sum` discussed previously, constraints, predefined functions, and the nested search operators `solve`, `minof`, and `maxof` that are presented later in this paper. Constraints are built using expressions and the traditional relational operators. Predefined functions are used to access the domains in a given computation state. Given an argument a , function `dsize(a)` returns the size of the domain of a ; `dmin(a)` returns the minimum value in the domain, `dmax(a)` the maximum value, `regretdmin(a)` the difference between the two smallest values (0 if only one exists in the domain), `regretdmax(a)` the difference between the two largest values (0 if only one exists); `dmid(a)` returns the value in the middle of the domain, while `bound(a)` return 1 if there is only one value left in the domain and 0 otherwise. The parameters used in the aggregate

$\langle Expr \rangle$	→ Integer
	→ $\langle Composite \rangle$
	→ $\langle Expr \rangle + \langle Expr \rangle$
	→ $\langle Expr \rangle * \langle Expr \rangle$
	→ $\langle Expr \rangle - \langle Expr \rangle$
	→ $\langle Expr \rangle / \langle Expr \rangle$
	→ $\text{abs}(\langle Expr \rangle)$
	→ $\text{sum}(\langle Parameter \rangle) \langle Expr \rangle$
	→ $(\langle Constraint \rangle)$
	→ $\langle Functions \rangle$
	→ $\text{solve}(\langle Choice \rangle)$
	→ $\text{minof}(\langle Expr \rangle, \langle Choice \rangle)$
	→ $\text{maxof}(\langle Expr \rangle, \langle Choice \rangle)$
$\langle Constraint \rangle$	→ $\langle Expr \rangle < \langle Expr \rangle$
	→ $\langle Expr \rangle \leq \langle Expr \rangle$
	→ $\langle Expr \rangle \geq \langle Expr \rangle$
	→ $\langle Expr \rangle > \langle Expr \rangle$
	→ $\langle Expr \rangle = \langle Expr \rangle$
	→ $\langle Expr \rangle \neq \langle Expr \rangle$
$\langle Functions \rangle$	→ $\text{dsize}(\langle Expr \rangle)$
	→ $\text{dmin}(\langle Expr \rangle)$
	→ $\text{dmax}(\langle Expr \rangle)$
	→ $\text{regretdmin}(\langle Expr \rangle)$
	→ $\text{regretdmax}(\langle Expr \rangle)$
	→ $\text{dmid}(\langle Expr \rangle)$
	→ $\text{bound}(\langle Expr \rangle)$
$\langle Parameter \rangle$	→ $\langle Identifier \rangle \text{ in } \langle Range \rangle$
$\langle Composite \rangle$	→ $\langle Identifier \rangle \langle Deref \rangle$
$\langle Deref \rangle$	→
	→ $[\langle expr \rangle^+] \langle Deref \rangle$
	→ $. \langle Identifier \rangle \langle Deref \rangle$
	→ $. \langle Identifier \rangle (\{ \langle expr \rangle \}) \langle Deref \rangle$

Fig. 4. The syntax of expressions and constraints.

operators are very simple in this paper: they simply consist of an identifier that takes its value in a range.

4. SEARCH IN OPL

This section reviews the main OPL constructs to support search in OPL. These constructs are used to specify the search tree to explore, though not how to explore it, which is the topic of Section 5. By default, OPL uses depth-first search

```

⟨Search⟩ → search ⟨Choice⟩ ;
⟨Choice⟩ →
  → try ⟨TrySeq⟩ endtry
  → tryall ( ⟨Parameter⟩ [ : ⟨Constraint⟩ ] [ ⟨Order⟩ ] ) ⟨Choice⟩
    [ onFailure ⟨Choice⟩ ]
  → forall ( ⟨Parameter⟩ [ : ⟨Constraint⟩ ] [ ⟨Order⟩ ] ) ⟨Choice⟩
  → select ( ⟨Parameter⟩ [ : ⟨Constraint⟩ ] [ ⟨Order⟩ ] ) ⟨Choice⟩
  → while ⟨Constraint⟩ do ⟨Choice⟩
  → if ⟨Constraint⟩ then ⟨Choice⟩ [ else ⟨Choice⟩ ] endif
  → let ⟨Id⟩ = ⟨Expr⟩ in ⟨Choice⟩
  → { ⟨Choice⟩+ }
  → when ⟨Constraint⟩ do ⟨Choice⟩
  → onValue ⟨Expr⟩ do ⟨Choice⟩
  → onRange ⟨Expr⟩ do ⟨Choice⟩
  → onDomain ⟨Expr⟩ do ⟨Choice⟩
  → ⟨Constraint⟩
  → ⟨PredefinedChoice⟩
  → ⟨Composite⟩ <- ⟨Exp⟩
  → ⟨Composite⟩ := ⟨Exp⟩
  → ⟨Strategy⟩

⟨TrySeq⟩ → ⟨Choice⟩
  → ⟨Choice⟩ | ⟨TrySeq⟩

⟨Order⟩ → ordered by increasing ⟨Exp⟩
  → ordered by decreasing ⟨Exp⟩
  → ordered by increasing < ⟨Expr⟩+ >
  → ordered by decreasing < ⟨Expr⟩+ >

```

Fig. 5. The syntax of search procedures.

(or depth-first branch and bound) for exploring search trees. This section starts by giving a high-level overview of the available constructs. It then illustrates them in detail, and shows their generality, through the implementation of typical constraint programming search techniques.

4.1 Overview

A search procedure in OPL starts with the keyword `search` followed by a choice. The choice instructions are mostly used to specify the search tree, i.e., an *ordered and/or tree* in artificial intelligence terminology. `try` and `tryall` are the two instructions of OPL for specifying (ordered) or-nodes, `try` having a fixed set of alternatives and `tryall` having as many alternatives as there are elements in its defining set. The sequencing operator `;` and instruction `forall` are the counterpart for specifying (ordered) and-nodes. In particular, the `forall` instruction generates as many conjuncts as there are elements in its defining set. The `select` instruction is a high-level construct to select an element from a set,

a functionality often useful in practice (e.g., choosing which machine to rank next in a scheduling problem). Of course, a choice can be a sequence of choices. Note also that, inside expressions, OPL offers the instructions `solve`, `minof`, and `maxof` for nested search.

OPL also offers traditional constructs for iterations (e.g., `while`) as well as conditionals. It is important to observe that expressions in conditional or iterative statements cannot contain variables (errors are raised when they contain variables), except in the predefined functions presented earlier in Section 3. These instructions only aim at controlling the execution flow.³

OPL also supports data-driven constructs introduced in constraint logic programming (e.g., [Dincbas et al. 1988]) and in concurrent constraint programming (e.g., [Maher 1987; Saraswat 1993]). The `when`, `onValue`, `onRange`, and `onDomain` instructions all execute their bodies when some condition is satisfied and postpone their execution otherwise.

The basic blocks to define the search tree are constraints, predefined choices, and assignments. OPL has a number of predefined search instructions (e.g., `generate`) that are not described in this paper (see [Van Hentenryck 1999] for a description). There are two types of assignments: *global* assignments (i.e., `:=`) that are visible to the whole tree and local assignments that are visible only on the branch on which they are executed.⁴

Finally, OPL supports the application of a, possibly user-defined, strategy to a choice. The search strategies are covered in the next section. We now present these search instructions in detail and illustrate them on typical search procedures.

4.2 The Standard Labeling Procedure

A search procedure in constraint programming often consists of assigning values to variables. Such a labeling procedure generally chooses which variable to instantiate next (variable choice) and then chooses, in a nondeterministic way, which value to assign to the selected variable (value choice). This process is repeated until all variables are instantiated. Note that the variable choice is deterministic (i.e., the choice of the variable is never reconsidered), while the value choice is, of course, nondeterministic.

Typically, standard labeling procedures are implemented in OPL using the `forall` and `tryall` instructions. The `forall` construct is used for iterating over the variables, while the `tryall` instruction is used to try the values in a nondeterministic way. Figure 6 extends the basic queens program with a simple labeling procedure that iterates over all the elements of `Domain` and, for each such element `i`, tries to assign, in a nondeterministic way, a value to `queen[i]` from `Domain`. It is important to point out, once again, that these instructions specify the search tree, not the way this search tree should be explored. In this context and using artificial intelligence terminology, a `forall` instruction defines an and-node, and a `tryall` instruction defines an or-node. Since the default strategy is depth-first search, the OPL implementation first selects the

³Note that logical implications and data-driven constructs can be used to state conditional constraints.

⁴Local assignments are often called reversible assignments in constraint programming, since their effect is undone upon backtracking.

```

int n = ...;
range Domain 1..n;
var Domain queen[Domain];
solve {
  forall(ordered i,j in Domain) {
    queen[i] <> queen[j];
    queen[i] + i <> queen[j] + j;
    queen[i] - i <> queen[j] - j
  };
};
search {
  forall(i in Domain)
  tryall(v in Domain)
  queen[i] = v;
};

```

Fig. 6. The basic n -queens program with a search procedure.

queen on the first column and places it on the first row. It then considers the second queen and places it on the third row, since the first two rows are attacked by the first queen. This process is iterated for all the variables. If, at any point, a variable cannot be given a value consistent with the constraint store, the OPL implementation goes back to the previous variable and tries another value.

4.2.1 Variable Ordering. It is often critical to choose carefully which variable to instantiate next, since this choice specifies the size and the shape of the search tree. The `forall` construct supports a fully dynamic variable choice. For instance, the search procedure

```

search {
  forall(i in Domain ordered by increasing dsize(queen[i]))
  tryall(v in Domain)
  queen[i] = v;
};

```

implements the so-called *first-fail* principle that consists of choosing first the variable with the smallest domain (i.e., the variable with the fewest possible values). In the search procedure, `dsize(queen[i])` returns the size of the domain in the current computation state. The procedure thus chooses the variable with the smallest domain and assigns a value to this variable nondeterministically. This assignment reduces the domains of the other variables through constraint propagation. The next iteration of the `forall` instruction chooses another variable, i.e., the one with the smallest domain in this new computation state, and the process is iterated until all variables have been instantiated (or no solution was found).

This search procedure for the queens problem can in fact be improved further by using a more advanced variable choice that illustrates the generality of orderings in OPL. The key idea is to choose the queen with the smallest domain and, in case of ties, the queen that is closest to the middle of the board. This

```

int n = ...;
range Domain 1..n;
enum Dim { col, row };
var Domain queen[Dim,Domain];
solve {
  forall(d in Dim & ordered i,j in Domain) {
    queen[d,i] <> queen[d,j];
    queen[d,i] + i <> queen[d,j] + j;
    queen[d,i] - i <> queen[d,j] - j
  };
  forall(i,v in Domain)
    queen[col,i] = v <=> queen[row,v] = i;
};
search {
  forall(i in Domain ordered by increasing <dsize(queen[col,i]),
    abs(n/2-i)>>)
    tryall(v in Domain ordered by increasing dsize(queen[row,i]))
      queen[col,i] = v;
};

```

Fig. 7. The stable n -queens program with a search procedure.

lexicographic ordering can be expressed naturally in OPL through the search procedure

```

search {
  forall(i in Domain ordered by increasing <dsize(queen[i]),abs(n/2-i)>>)
    tryall(v in Domain)
      queen[i] = v;
};

```

where function `abs` returns the absolute value of its argument.

4.2.2 Value Ordering. The other important aspect of a labeling procedure is the order in which values are tried. The `tryall` instruction also supports the definition of fully dynamic value ordering. Consider the program depicted in Figure 7 that extends the redundant modeling program with a search procedure. Its search procedure uses the first-fail principle at two levels: to choose the next variable to instantiate in the `forall` instruction and to choose which value to assign in the `tryall` instruction. Intuitively, a step in this search procedure consists of

- (1) choosing a column variable to instantiate next using the same criteria as before;
- (2) instantiating this column variable by trying first the values whose corresponding row variable has the smallest domain.

Observe the conciseness, simplicity, and elegance of this search procedure.

It is also interesting to note that this labeling procedure does not exploit the fact that, when an assignment, say `queen[col, i] = 3`, fails, the additional

constraint `queen[col,i] <> 3` can be added to the constraint store. This behavior could be obtained in OPL by writing

```
search {
  forall(i in Domain
    ordered by increasing <dsize(queen[col,i]),abs(n/2-i)>)
    tryall(v in Domain ordered by increasing dsize(queen[row,i]))
      queen[col,i] = v
    onFailure
      queen[col,i] <> v;
};
```

The intuition is that, when an assignment `queen[col,i] = v` is reconsidered, the OPL implementation adds constraint `queen[col,i] <> v` to the constraint store before considering any alternative choice in the `tryall` instruction.⁵ It is useful to mention that the functionality provided by the `onFailure` can be simulated by a `while` loop and a `try` instruction (as shown later in the paper), but the price to pay is a loss in readability.

This form of labeling, combining both a variable and a value ordering, is frequent in constraint programming. For instance, consider the warehouse location program presented in Figure 3. A well-known heuristic for this problem is the so-called *maximal regret*. To understand this heuristic intuitively, consider what happens when a given store is not assigned to its cheapest supplier. If its second cheapest supplier has roughly the same cost, then not selecting the cheapest customer does not induce a substantial loss. However, if the next cheapest supplier is much more expensive, the penalty is more substantial. The maximal regret heuristic exploits this observation. At any given time of the search, there are a number of stores whose suppliers remain to be determined and a number of warehouses that can be selected. The *regret* of a store in this computation state is the difference between its first and second choice and the *maximal regret* heuristic recommends

- to select the supplier with the maximal regret;
- to try the warehouses in increasing order of cost.

The search procedure

```
search {
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};
```

is an implementation in OPL featuring dynamic variable and value orderings.

It is also interesting at this point to mention that search procedures in OPL do not need to assign values to all variables. For instance, the search procedure in the warehouse location program does not assign a value to the variables in array `open`. In these cases, OPL applies its default search procedure to instantiate the

⁵Of course, in this simple case, the system could deduce this information automatically. There are more complex applications where this is not possible, since the `tryall` instruction applies to an arbitrary search procedure. This is discussed later in this paper.

remaining variables. This design choice is motivated by the fact that OPL is a modeling language, and modeling languages typically do not require, or even allow, users to specify search procedures.

4.2.3 Variations on the Standard Labeling Procedure. It is also possible to write labeling procedures in different ways. For instance, the search procedure

```
search {
  forall(i in Domain)
    tryall(v in Domain)
      queen[i] = v
    onFailure
      queen[i] <> v;
};
```

could be rewritten as

```
search {
  forall(i in Domain)
    while not bound(queen[i]) do
      let v = dmin(queen[i]) in
        try
          queen[i] = v | queen[i] <> v
        endtry;
};
```

The body of the while instruction (that assigns or removes a value for a queen) is iterated until the variable is bound. The while instruction, combined with the bound function, is particularly useful for value-directed labeling to be discussed shortly.

4.2.4 Domain Splitting. In some applications, it is preferable to split the domain of a variable in several parts instead of assigning values directly (see, e.g., [Dincbas et al. 1988]). The search procedure

```
search {
  forall(i in Domain)
    while not bound(queen[i]) do
      let m = dmid(queen[i]) in
        try
          queen[i] <= m | queen[i] > m
        endtry;
};
```

illustrates domain-splitting. Function `dmid` returns the value in the middle of the domain of its argument. It is of course possible to combine domain-splitting with variable orderings and to use conditional expressions to choose which part of the domain to explore first. In mixed integer programming, it is also possible in OPL to use the value of the variable in the current linear relaxation of the constraints. See [Van Hentenryck 1999] for more details on how to use this information.

4.3 Value-Directed Labeling

Some search procedures start by choosing first the value to assign and then select which variable to instantiate. These search procedures are called value-directed labelings in this paper. Consider the queens problem modeled in Figure 1 again, and assume that the following heuristic must be implemented: select the value that corresponds to the noninstantiated variable with the smallest domain and assigns this value (in a nondeterministic way) to the column variable with the smallest domain and closest to the middle of the chessboard. The search procedure

```

search {
  while not bound(queen) do
    select(v in Domain: not bound(queen[row,v])
          ordered by increasing dsize(queen[row,v]))
    tryall(i in Domain
          ordered by increasing <dsize(queen[col,i]),abs(n/2-i)>)
      queen[col,i] = v
    onFailure
      queen[col,i] <> v;
};

```

is an implementation of this heuristic. Typically, value-directed labelings are programmed by a combination of a while loop, a select instruction, and try or tryall instructions. The select instruction is a high-level construct to select an element from a set that satisfies a condition. In the example, the select instruction selects the value v , i.e., the index of the row variable that is not bound and has the smallest domain. The tryall instruction chooses the index of the column variable to instantiate, since it is known that one of the queens must take this value. The process is repeated until all variables are bound.

4.4 Nested Search

In some computationally hard applications, it is useful to collect additional information during the search by trying some hypotheses and analyzing their effects. The collected information can then be used to prune the search space or to guide the search process heuristically. The hypotheses are tested by performing a nested search, i.e., a search inside the search procedure. The nested search defines and manages its own search tree, leaves the computation state unchanged, and returns the appropriate information.⁶

4.4.1 Nested Search for Pruning. To prune the search space, nested search can try out some hypothesis (e.g., a constraint). If the nested search fails, the negation of the hypothesis is added to the constraint store. If it succeeds, no conclusion can be drawn. Instruction `solve` can be used to perform a nested search. More precisely, if p is a search procedure and if σ is the current constraint store, `solve(p)` returns true if and only if there is a solution of p that is consistent with σ . To illustrate nested search, consider the following search procedure for the queens problem

⁶Note that nested search can be implemented in logic programming by a double negation as failure, possibly using `assert` to store the relevant information. It is also related to probing in mathematical programming [Savelbergh 1994].

```

search {
  forall(i in Domain
    ordered by increasing <dsize(queen[col,i]),abs(n/2-i)>) {
    forall(v in Domain)
      if not solve(queen[col,i] = v) then
        queen[col,i] <> v;
      endif;
    tryall(v in Domain ordered by increasing dsize(queen[row,i]))
      queen[col,i] = v
    onFailure
      queen[col,i] <> v;
  };
};

```

and the instructions

```

forall(v in Domain)
  if not solve(queen[col,i] = v) then
    queen[col,i] <> v;
  endif;

```

The basic idea here is to consider the column variable i and to try reducing its domain by testing whether it can be assigned specific values. The function `solve(queen[col,i] = v)` performs the nested search. It returns true if adding `queen[col,i] = v` to the constraint store does not cause a failure, and it returns false otherwise. When the nested search returns false, the corresponding value can be removed from the domain. Note again that the nested search has no effect on the search tree. It creates an independent subtree that is thrown away when the nested search returns. In this example, the tree consists of a single node (i.e., the additional constraint), but nested search can be applied on arbitrary search procedures as shown in the next subsection.

Similar instructions can be used to tighten the bounds of the domains (instead of the entire domain), e.g.,

```

while not solve(queen[col,i] = dmin(queen[col,i])) do
  queen[col,i] <> dmin(queen[col,i]);
while not solve(queen[col,i] = dmax(queen[col,i])) do
  queen[col,i] <> dmax(queen[col,i]);

```

It is interesting to point out that the function `solve` can be applied to any OPL search procedure.

4.4.2 Nested Search for Heuristics. Nested search can also be used to collect heuristic information. For optimization problems, the basic idea consists of trying out an hypothesis and to evaluate its effect on the objective value. It is thus possible to choose, for instance, the hypothesis leading to the smallest increase in the minimum value of the objective function. To illustrate this idea, consider the warehouse location problem again and the heuristic that consists of choosing the assignment of a warehouse to a supplier that minimizes the increase in the objective function. The search procedure

```

search {
  forall(s in Stores ordered by decreasing
        minof(totalCost,tryall(w in Warehouses) supplier[s] = w))
    tryall(w in Warehouses ordered by increasing
          minof(totalCost,supplier[s] = w)
          supplier[s] = w;
};

```

implements this heuristic. Both the `forall` and the `tryall` instructions use `minof` function that performs the nested search. More precisely, the function `minof(f, c)` performs a nested search of `c` and returns the smallest value of `f` in the computation states of the leaves of the tree defined by `c`.

Note that the nested search is a simple equation for the `tryall` instruction, while it involves a more complex search procedure in the case of the `forall` construct.

4.5 Demons

OPL also supports a number of data- or constraint-driven constructs that are useful in a variety of applications and are often called demons. Consider the instruction

```

forall(i,v in Domain)
  queen[col,i] = v <=> queen[row,v] = i;

```

from Figure 7. It could be replaced by inserting the instructions

```

forall(i in Domain)
  forall(v in Domain) {
    when queen[col,i] <> v do queen[row,v] <> i;
    when queen[row,v] <> i do queen[col,i] <> v;
  }
};

```

at the beginning of the search procedure. The basic idea underlying a `when` instruction is to execute its body as soon as its condition is entailed by the constraint store. In the above example, as soon as the domain of variable `queen[col,i]` does not contain `v`, the OPL implementation adds `queen[row,v] <> i` to the constraint store.

OPL includes other data-driven constructs, e.g., `onValue`, `onRange`, and `onDomain`, that execute a search procedure when their first argument (an expression) is given a value (`onValue`) or when its bounds (`onRange`) or its domain (`onDomain`) are updated. The `onRange` instruction is illustrated later in this paper.

4.6 Destructive Assignments

OPL also supports destructive assignments that are sometimes needed for complex applications. Consider the warehouse location problem of Figure 3 again and the search procedure

```

forall(s in Stores ordered by decreasing regretdmin(cost[s]))
  tryall(w in Warehouses ordered by increasing supplyCost[s,w])
    supplier[s] = w;

```

```

int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores 0..nbStores-1;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores & w in Warehouses) supplyCost[s,w];

var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;
var int totalCost in 0..maxint;
int freq[w in Warehouses] = 0;

minimize
  totalCost
subject to {
  totalCost = sum(s in Stores) cost[s] + sum(w in Warehouses)
    fixed * open[w];
  forall(s in Stores)
    cost[s] = supplyCost[s,supplier[s]];
  forall(s in Stores)
    open[supplier[s]] = 1;
  forall(w in Warehouses)
    sum(s in Stores) (supplier[s] = w) <= capacity[w];
};
search {
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing <supplyCost[s,w],
      freq[w]>) {
      supplier[s] = w;
      freq[w] <- freq[w] + 1;
    };
};

```

Fig. 8. The warehouse-location program with destructive assignments.

that assigns the warehouses to the selected stores in increasing order of the supply costs. Assume that ties must be broken by choosing the warehouses currently used by the smallest number of stores. Figure 8 depicts an OPL program implementing this idea. The program declares an array of frequencies

```
int freq[w in Warehouses] = 0;
```

that are initialized to 0, and the search procedure becomes

```

forall(s in Stores ordered by decreasing regretdmin(cost[s]))
  tryall(w in Warehouses
    ordered by increasing <supplyCost[s,w],freq[w]>) {
    supplier[s] = w;
    freq[w] <- freq[w] + 1;
  };

```

This search procedure uses a lexicographic criterion used to select the warehouse to assign and a local assignment $\text{freq}[w] \leftarrow \text{freq}[w] + 1$ to increase the frequency of w . Note that the assignment is only visible on the branch of the tree where it takes place (e.g., if a depth-first strategy is used, the assignment is undone upon backtracking, and the previous value is restored).

5. STRATEGIES IN OPL

The search instructions presented earlier specify a partially ordered search tree to be explored by the OPL implementation. They do not, however, specify how to explore this tree. The purpose of this section is to describe the support in OPL to control the exploration of the search tree. The section is divided in three parts: search strategies, search limits, and search selectors. Search strategies specify how to explore search trees; search limits specify when to stop searching, while search selectors specify which leaves of the search tree should be considered. Figure 9 describes the relevant syntax for this paper.

$\langle \textit{Strategy} \rangle$	→ $\langle \textit{SearchStrategy} \rangle$ → $\langle \textit{SearchLimit} \rangle$ → $\langle \textit{SearchSelector} \rangle$
$\langle \textit{SearchStrategy} \rangle$	→ $\text{LDSearch} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{BFSearch} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{applyStrategy} (\textit{Constructor}) \langle \textit{Choice} \rangle$
$\langle \textit{SearchLimit} \rangle$	→ $\text{timeLimit} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{failLimit} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{applyLimit} (\textit{Constructor}) \langle \textit{Choice} \rangle$
$\langle \textit{SearchSelector} \rangle$	→ $\text{minimize} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{maximize} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$ → $\text{firstSolution} (\langle \textit{Expr} \rangle) \langle \textit{Choice} \rangle$
$\langle \textit{Constructor} \rangle$	→ $\langle \textit{Identifier} \rangle (\langle \textit{expr} \rangle^+)$
$\langle \textit{SearchStrategyDecl} \rangle$	→ $\text{SearchStrategy} (\textit{Identifier}) (\{ \underline{\langle \textit{Arg} \rangle} \}) \{$ $\text{evaluated to } \langle \textit{Expr} \rangle ;$ $\text{postponed when } \langle \textit{Expr} \rangle ;$ $\}$
$\langle \textit{SearchLimitDecl} \rangle$	→ $\text{SearchLimit} (\textit{Identifier}) (\{ \underline{\langle \textit{Arg} \rangle} \})$ $\text{when } \langle \textit{Expr} \rangle ;$
$\langle \textit{Arg} \rangle$	→ $\langle \textit{Type} \rangle \langle \textit{Identifier} \rangle$

Fig. 9. The syntax of search strategies.

5.1 Exploration Strategies

As mentioned, the search instructions presented so far describe the search tree to explore but do not specify how to explore it. By default, OPL uses depth-first search to explore the specified search tree. In addition, it provides a number of predefined exploration strategies, including best-first search and limited discrepancy search [Harvey and Ginsberg 1995], that may be applied to any search procedure defined in OPL. Finally, OPL makes it possible to define new exploration strategies at a very-high level of abstraction. Section 5.1.1 describes how to use the predefined strategies. Section 5.1.2 describes the underlying computational model, and Section 5.1.3 illustrates how to define new strategies.

5.1.1 Predefined Exploration Strategies. Best-first search assumes the existence of an objective function that assigns a numerical value to any or-node of the search tree. It explores the search tree by always selecting the or-node with the smallest objective value as the node to be expanded next. The search procedure

```
search {
  BFSearch(totalCost)
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};
```

illustrates how to apply best-first search to the warehouse location problem. The instruction `BFSearch` specifies that the search tree specified by the instructions

```
forall(s in Stores ordered by decreasing regretdmin(cost[s]))
  tryall(w in Warehouses ordered by increasing supplyCost[s,w])
    supplier[s] = w;
```

must be explored by using best-first search using the minimal value `totalCost` as objective function. In other words, the node in the search tree associated with the smallest value of `dmin(totalCost)` is selected first to be expanded.

Limited discrepancy search (LDS) [Harvey and Ginsberg 1995] is a strategy that is effective when a good heuristic is available. It consists in exploring the search tree in waves, each wave allowing more discrepancies with respect to the heuristics. The first wave simply follows the heuristics. Assuming that the search tree is a binary tree (all OPL search trees are transformed into binary search trees), this means that first wave simply follows the left-most path of the tree. The second wave explores the part of the search tree obtained by assuming that the heuristic makes one mistake (or discrepancy), i.e., it takes exactly one right branch somewhere along the path from the root to a leaf. Subsequent waves explore the parts of tree obtained by assuming 2, 3, ... discrepancies with respect to the heuristic. By trusting the heuristic, LDS may reach good solutions (and thus the optimal solution) much faster than depth-first and best-first search on some applications. Of course, LDS may be generalized to allow for larger increases in the numbers of discrepancies inside waves. For instance, the search procedure

```

search {
  LDSearch(4)
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};

```

uses LDS with discrepancy increments of 4 on the warehouse location problem.

5.1.2 The Computation Model. Before presenting how to define an exploration strategy in OPL, it is important to describe the computation model that underlies both predefined and user-defined strategies. The computation model is presented in terms of an exploration algorithm based on three basic principles. First, the exploration algorithm only works on the or-nodes (or choice points) of the search tree, which is assumed to be binary. Second, it assumes the existence of an evaluation function that assigns a numerical value with each node. Third, the algorithm has a procedure to decide when to postpone a node and to switch to the node with the best evaluation. The exploration algorithm only assumes a small set of operations on nodes. They include the ability to test whether a node is a failure node (i.e., a node whose associated constraint store is inconsistent), a leaf node, or an internal node, the ability to get the left and the right children of an internal node, and the ability to obtain the evaluation of a node.

Figure 10 is a high-level description of the exploration algorithm. The algorithm is organized around two concepts: a priority queue that maintains the nodes according to their evaluations and an active node. It consists of two mutually recursive procedures. Procedure `explore` is the entry point: it should be called initially with a priority queue containing the top-level or-node, and it returns true if and only if a solution is found. Its implementation simply tests whether the queue is empty, in which case no solution has been found. Otherwise, the procedure pops the node with the best evaluation to become the active node and calls procedure `exploreActiveNode`. This procedure makes a case analysis on the active node `a`. If `a` is a failure node, then procedure `explore` is called recursively with a smaller priority queue. If `a` is a leaf node, then a solution has been found. If `a` must be postponed, which means that the algorithm has decided to switch to the node with best evaluation, the active node is inserted in the queue and procedure `explore` is called recursively. Otherwise, the active node is expanded: the right child is inserted in the priority queue; the left child becomes the active node; and procedure `exploreActiveNode` is called recursively.

It is easy to see that the predefined exploration strategies of OPL are instances of this generic exploration algorithm. Best-first search uses its objective function as the evaluation function. Limited discrepancy search uses the number of times a right branch is taken as the evaluation function and thus explores first those nodes with fewer discrepancies. Depth-first search uses minus the depth of the node as evaluation function.⁷ In these three exploration strategies, it also suffices to postpone a node as soon as a node with a better evaluation is available (which, in fact, only occurs in best-first search).

⁷Note also that the algorithm always prefers left branches, since right children are always inserted in the priority queue.

```

Boolean explore(PriorityQueue Q)
{
  if Q.empty() {
    return false;
  } else {
    Node a := Q.pop();
    return exploreActiveNode(a,Q);
  }
}

Boolean exploreActiveNode(Node a, PriorityQueue Q)
{
  if a.isFailNode() {
    return explore(Q);
  } else if a.isLeafNode() {
    return true;
  } else if a.mustBePostponed(Q) {
    Q.insert(a);
    return explore(Q);
  } else {
    Q.insert(a.getRight());
    a := a.getLeft();
    return exploreActiveNode(a,Q);
  }
}

```

Fig. 10. The exploration algorithm of OPL.

5.1.3 *User-Defined Exploration Strategies.* Now that the computational model has been presented, it is easy to see that defining a novel search strategy in OPL consists of specifying

- (1) an expression specifying the evaluation function of a node;
- (2) a boolean expression that specifies when to postpone the current node in order to switch to the node with the best evaluation function.

These expressions are evaluated in the computation state of the active node, and OPL gives access to a variety of information on the active and best nodes. Let us illustrate how exploration strategies can be defined in OPL by showing how to declare the predefined search strategies. The instruction

```

SearchStrategy myBFS(var int obj) {
  evaluated to dmin(obj);
  postponed when
    OplSystem.getEvaluation() > OplSystem.getBestEvaluation();
};

```

defines a best-first search strategy. The evaluation function `dmin(obj)` is given as the smallest value in the domain of the objective function. A node is postponed when its evaluation (given by the OPL system call `OplSystem`.

`getEvaluation()` is worse than the best evaluation value (given by `OplSystem.getBestEvaluation()`). This strategy can be easily generalized by allowing a tolerance before postponing the active node as in the instruction

```
SearchStrategy myBFS(var int obj,int n) {
    evaluated to dmin(obj);
    postponed when
        OplSystem.getEvaluation() > OplSystem.getBestEvaluation() + n;
};
```

which adds the tolerance `n` to the best evaluation value. Consider now the instruction

```
SearchStrategy myLDS(int n) {
    evaluated to OplSystem.getRightDepth();
    postponed when
        OplSystem.getEvaluation() > OplSystem.getBestEvaluation() + n;
};
```

that specifies an LDS strategy. Its evaluation function uses the OPL system call `OplSystem.getRightDepth()` that returns the number of right branches taken to reach the active node. In addition, the active node is postponed when its number of discrepancies is greater than the smallest number of discrepancies of the best node in the queue plus the tolerance `n`. Finally, consider the instruction

```
SearchStrategy myDFS() {
    evaluated to - OplSystem.getDepth();
    postponed when
        OplSystem.getEvaluation() > OplSystem.getBestEvaluation();
};
```

that specifies an depth-first search strategy. Its evaluation is simply minus the depth of the node.

Once an exploration strategy is declared, it is easy to apply by using instructions of the form

```
applyStrategy myBFS(totalCost,4)
    forall(s in Stores ordered by decreasing regretdmin(cost[s]))
        tryall(w in Warehouses ordered by increasing supplyCost[s,w])
            supplier[s] = w;
```

that applies a best-first search exploration strategy (with a tolerance of 4) to the search procedure of the warehouse location. Figure 11 depicts a complete OPL programming for the warehouse location problem with a user-defined strategy.

5.2 Limit Strategies

On many hard problems, it is impossible to find an optimal solution within reasonable resources. OPL makes it possible to use incomplete search procedures by limiting the resources (e.g., computation time) used during search. For instance, the instruction

```

int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores 0..nbStores-1;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores & w in Warehouses) supplyCost[s,w];

var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;
var int totalCost in 0..maxint;

SearchStrategy myBFS(var int obj,int n) {
  evaluated to dmin(obj);
  postponed when OplSystem.getEvaluation() > OplSystem.
    getBestEvaluation() + n;
};

minimize
  totalCost
subject to {
  totalCost = sum(s in Stores) cost[s] + sum(w in Warehouses)
    fixed * open[w];
  forall(s in Stores)
    cost[s] = supplyCost[s,supplier[s]];
  forall(s in Stores)
    open[supplier[s]] = 1;
  forall(w in Warehouses)
    sum(s in Stores) (supplier[s] = w) <= capacity[w];
};

search {
  applyStrategy myBFS(totalCost,4)
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};

```

Fig. 11. The warehouse-location program with a user-defined strategy.

```

search {
  timeLimit(60)
  applyStrategy myLDS(4)
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};

```

applies LDS to the search procedure of the warehouse location problem for 60 seconds. It returns the best solution found in this time frame (if any).

It is easy to upgrade the computation model shown previously to integrate limit strategies. The key idea is to specify a condition that specifies when the search must be aborted. As a consequence, it is also a simple matter to let users define their own search limits. For instance, the instruction

```
SearchLimit cutoff(int nb)
  when OplSystem.getNumberOfFails() > nb;
```

declares a search limit that aborts the search when the number of fails exceed a certain threshold. It can be applied as follows

```
search {
  applyLimit cutoff(1000)
  applyStrategy myLDS(4)
  forall(s in Stores ordered by decreasing regretdmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};
```

to limit the search in the warehouse location problem to no more than 1000 failures.

5.3 Selection Strategies

This section has presented two types of strategies so far: exploration strategies that specify how to explore a search tree and limit strategies that specify when to abort the search. The last type of strategies available in OPL are selection strategies. They specify which leaves of a search tree should be considered as solutions. There are two main kinds of selection strategies available in OPL: those that return a solution that minimizes or maximizes an objective function and those who return the first n solutions. For instance, it is possible to write a search procedure of the form

```
search {
  LDSearch(4) minimize(cost) {
    ...
  };
  maximize(fairness) {
    ...
  };
}
```

that first minimizes cost using LDS and then maximizes fairness using depth-first search.

Finally, there are many circumstances where a search procedure should produce only a single solution or a couple of solutions. For instance, the search procedure

```
search {
  firstSolution(3)
  forall(i in D)
    generate(queen[i]);
};
```

returns the first 3 solutions to the queens problem.

6. SCHEDULING

This section illustrates that the search facilities previously described are instrumental in capturing common search procedures in the scheduling area. It discusses ranking in job-shop scheduling, time-directed labeling, and the technique of shaving. OPL has specific support for scheduling applications given the practical importance of this class of applications. Once again, it is outside the scope of this paper to describe them in detail. It suffices here to mention that it supports the concepts of activities and resources. For the purpose of this paper, an activity can be thought of as composed of three variables, i.e., its starting date, its duration, and its ending date, together with the constraints linking them. A unary resource is a resource that cannot be shared by two activities at any time, i.e., two activities requiring the same unary resource cannot be scheduled at the same time. By default, the OPL implementation uses the edge-finder algorithm [Carlier and Pinson 1990] on unary resources for pruning the search space. A discrete resource is a resource that can be shared by several resources. It has a limited capacity however, and the total demand of the activities using the discrete resource at a given time cannot exceed the capacity.

6.1 Ranking

Ranking is the counterpart of labeling for job-shop scheduling. Consider a typical job-shop scheduling application that schedules a set of tasks subject to precedence and disjunctive constraints. The problem consists of a set of jobs, each job being a sequence of tasks. Each task requires a resource during its execution, and two tasks sharing the same resource cannot overlap in time. The goal of the program is to minimize the makespan, i.e., the earliest completion time of the project. Figure 12 depicts an OPL program for the job-shop problem. It first declares the number of machines, the number of jobs, and the number of tasks in the jobs. The main data of the problem, i.e., the duration of all the tasks and the resources they require, are then given. The next two instructions are useless for the program at this point but will be important to illustrate some search procedures subsequently. The instructions

```
Activity task[j in Jobs, t in Tasks](duration[j,t]);
Activity makespan(0);

UnaryResource tool[Machines];
```

declare the activities and the unary resources. Note that the makespan is modeled for simplicity as an activity of duration zero. The first set of constraints specifies that the makespan is not smaller than the ending time of the last task of each job. The next two sets specify the precedence and disjunctive constraints.

As mentioned, in job-shop scheduling, the main issue is to rank the tasks on the machines, i.e., to determine the order in which the tasks must be executed. Once the tasks are ranked, it is easy to find the earliest starting date for the tasks respecting the ranking. A typical search procedure for job-shop scheduling then consists of selecting a unary resource and ranking its activities. It is also important to rank first the machine that seems the tightest in the computation state. Here is a search procedure based on these principles:

```

int nbMachines = ...;
range Machines 1..nbMachines;
int nbJobs = ...;
range Jobs 1..nbJobs;
int nbTasks = ...;
range Tasks 1..nbTasks;

Machines resource[Jobs,Tasks] = ...;
int+ duration[Jobs,Tasks] = ...;
int totalDuration = sum(j in Jobs, t in Tasks) duration[j,t];
struct Task { Jobs j; Tasks t; };
{Task} tasks[m in Machines] = {<j,t>| j in Jobs & t in Tasks:
    resource[j,t] = m};

ScheduleHorizon = totalDuration;
Activity task[j in Jobs, t in Tasks](duration[j,t]);
Activity makespan(0);

UnaryResource tool[Machines];

minimize
    makespan.end
subject to {
    forall(j in Jobs)
        task[j,nbTasks] precedes makespan;

    forall(j in Jobs)
        forall(t in 1..nbTasks-1)
            task[j,t] precedes task[j,t+1];

    forall(j in Jobs)
        forall(t in Tasks)
            task[j,t] requires tool[resource[j,t]];
};

```

Fig. 12. A job-shop scheduling program.

```

search {
    while not isRanked(tool) do
        select(m in Machines: not isRanked(tool[m])
            ordered by increasing localSlack(tool[m]))
            rank(tool[m]);
};

```

In this search procedure, function `localSlack` returns a measure of the tightness of a unary resource, while `rank` is a nondeterministic procedure to rank the selected resource. Ranking a resource can be implemented as follows:

```

while not isRanked(tool[m]) do
    select(<j,t> in tasks[m]: isPossibleFirst(tool[m],task[j,t])
        ordered by increasing dmin(task[j,t].start))
        tryRankFirst(tool[m],task[j,t]);
};

```

These instructions select an activity using the selected resource and try to schedule it first. Function `isPossibleFirst(u, a)` returns true if activity `a` can possibly be scheduled first on unary resource `u` in the current computation state. Procedure `tryRankFirst(u, a)` is nondeterministic and has two alternatives: the case when `a` is scheduled before all unranked activities on `u` and the case when `a` is not ranked before these activities. It could be implemented as follows:

```
try
  rankFirst(tool[m], task[j, t]) | rankNotFirst(tool[m], task[j, t])
endtry;
```

where `rankFirst` specifies that the task must precede all other unranked activities on the resource, while `rankNotFirst` specifies that the task must be preceded by at least one of the unranked activities. There are of course many possible ranking procedures. For instance, the ranking procedure

```
search {
  while not isRanked(tool) do
    select(m in Machines: not isRanked(tool[m])
      ordered by increasing localSlack(tool[m]))
    select(<j, t> in tasks[m] : isPossibleFirst(tool[m])
      ordered by increasing dmin(task[j, t].start))
    tryRankFirst(tool[m], task[j, t]);
  };
```

reconsiders the selection of the machine to rank after each ranking decision, instead of ranking machines entirely as before. This is achieved by two nested `select` instructions that select respectively a machine and then a task to rank on the machine. The conciseness of both ranking procedures, that are in fact rather different, is a good example of the expressiveness of OPL. It is also useful to observe that both the search and the data-modeling facilities of OPL play a role in this conciseness.

In job-shop scheduling, it is often useful to apply LDS to the ranking procedure. For instance, here is a simple OPL search procedure

```
search {
  LDSearch()
  while not isRanked(tool) do
    select(m in Machines: not isRanked(tool[m])
      ordered by increasing localSlack(tool[m]))
    rank(tool[m]);
  };
```

that solves MT10 [Carlier and Pinson 1990] in about 90 seconds on a 300MHz Pentium PC.

6.2 Nested Search

Nested search is instrumental to implement the concept of shaving proposed in [Martin and Shmoys 1996]. The key idea here is to use nested search to determine which activities can be ranked first (or last or both). The search procedure

```

search {
  while not isRanked(tool) do {
    forall(m in Machines: not isRanked(tool[m]))
      forall(<j,t> in tasks[m]: isPossibleFirst(tool[m],task[j,a]))
        if not solve(rankFirst(tool[m],task[j,t])) then
          rankNotFirst(tool[m],task[j,t]);
        endif;
    select(m in Machines: not isRanked(tool[m])
      ordered by increasing localSlack(tool[m]))
      rank(tool[m]);
  };

```

shows how to implement shaving in OPL for unary resources. The instructions to study are

```

forall(<j,t> in tasks[m]: isPossibleFirst(tool[m],task[j,a]))
  if not solve(rankFirst(tool[m],task[j,t])) then
    rankNotFirst(tool[m],task[j,t]);
  endif;

```

whose idea is to iterate over all activities that can possibly be ranked first on a resource (among all nonranked activities) and to try to rank them first using the nested search

```

solve(rankFirst(tool[m],task[j,t]))

```

If the nested search fails, then the search procedure adds the constraint that at least one of the nonranked activities must be scheduled before the considered activity, i.e., it executes the instruction

```

rankNotFirst(tool[m],task[j,t]);

```

6.3 Time-Directed Labeling

When a resource is available in multiple units, ranking no longer applies, and other search procedures are needed. Consider the traditional ship-loading problem from [Roseaux 1985]. It consists of 34 activities subject to precedence constraints. In addition, all of these activities have durations and require a unique resource of capacity 8 in various quantities. For instance, activity 1 has a duration of 3 and requests 4 units of the discrete resource. The goal of the application is to minimize the makespan (i.e., the time to load the ship) while satisfying the precedence and capacity constraints. Figure 13 gives an OPL program (from [Van Hentenryck 1999]) for solving this problem. It is interesting to review some of the functionalities of OPL used in this program. First, the declaration

```

DiscreteResource res(8);

```

declares a discrete resource of capacity 8 for the cranes, and the OPL implementation makes sure that, at any time t , the demand of the activities requiring the resource at t is not greater than the capacity (i.e., 8 in this example) and that any activity whose demand would violate this constraint is not scheduled at time t . Second, the constraint

```

int capacity = 8;
int nbTasks = 34;
range Tasks 1..nbTasks;
int duration[Tasks] = [
    3, 4, 4, 6, 5, 2, 3, 4, 3, 2, 3, 2, 1, 5, 2, 3, 2, 2, 1, 1,
    1, 2, 4, 5, 2, 1, 1, 2, 1, 3, 2, 1, 2, 2];
int totalDuration = sum(t in Tasks) duration[t];
int demand[Tasks] = [
    4, 4, 3, 4, 5, 5, 4, 3, 4, 8, 4, 5, 4, 3, 3, 3, 6, 7, 4, 4,
    4, 4, 7, 8, 8, 3, 3, 6, 8, 3, 3, 3, 3, 3];
struct Precedences {
    int before;
    int after;
};
{Precedences} setOfPrecedences = {
    <1,2>, <1,4>, <2,3>, <3,5>, <3,7>, <4,5>, <5,6>, <6,8>, <7,8>, <8,9>,
    <9,10>, <9,14>, <10,11>, <10,12>, <11,13>, <12,13>, <13,15>, <13,16>,
    <14,15>, <15,18>, <16,17>, <17,18>, <18,19>, <18,20>, <18,21>, <19,23>,
    <20,23>, <21,22>, <22,23>, <23,24>, <24,25>, <25,26>, <25,30>, <25,31>,
    <25,32>, <26,27>, <27,28>, <28,29>, <30,28>, <31,28>, <32,33>, <33,34> };

scheduleHorizon = totalDuration;
Activity a[t in Tasks](duration[t]);
DiscreteResource res(8);
Activity makespan(0);
minimize
    makespan.end
subject to {
    forall(t in Tasks)
        a[t] precedes makespan;
    forall(p in setOfPrecedences)
        a[p.before] precedes a[p.after];
    forall(t in Tasks)
        a[t] requires(demand[t]) res;
};

```

Fig. 13. The ship-loading program.

```

forall(t in Tasks)
    a[t] requires(demand[t]) res;

```

specifies that activity $a[t]$ requires $\text{demand}[t]$ units of the discrete resource. For this application, OPL returns an optimal solution of the form

Optimal Solution with Objective Value: 66

```

a[1] = [0 -- 3 --> 3]
a[2] = [3 -- 4 --> 7]
a[3] = [7 -- 4 --> 11]
a[4] = [3 -- 6 --> 9]
...

```

Observe that, at time 3, activities $a[2]$ and $a[4]$ both require 4 units of the resource, implying that no other activity can be scheduled at this time.

An appropriate search strategy for this program is a time-directed labeling, i.e., a value-directed labeling where values represent starting times. The basic idea is to consider the earliest time t at which a nonscheduled activity can be started and to try to schedule activities at time t . The search procedure

```

search {
  while not bound(a) do
    select(t in Tasks: not bound(a[t].start)
      ordered by increasing dmin(a[t].start))
    let time = dmin(a[t].start) in
    forall(t in Tasks:
      not bound(a[t].start) & dmin(a[t].start) = time)
      try
        a[t].start = time
      |
        a[t].start > time
      endtry;
};

```

selects this earliest time $time$ (using the `select` and `let` instructions) and considers all activities that can be scheduled at $time$. For each of them, the search procedure decides whether to schedule it at time $time$ or later. This form of value-directed labeling is often useful in scheduling applications.

6.4 Dominance Relations

It is possible to improve the search procedure for the ship-loading problem by using the dominance relations of [Demeulesmeester and Herroelen 1992]. The basic observation behind dominance relations is as follows: if an activity a was not selected at time t (although it could have been), then it must be the case, in an optimal solution, that another activity will constrain a to be scheduled at a later date. A search procedure can exploit this observation to prune the search space as follows:

- (1) When an activity is not selected (and could have been) to be scheduled at time t , it is marked as postponed. It remains postponed until its range changes.
- (2) If, at any point, the current earliest starting time of the activities that are neither scheduled nor postponed is greater or equal to the latest starting date or the earliest finishing date of a postponed activity, then it is known that there exists a solution where the postponed activity is scheduled earlier and that the search procedure may fail.

A search procedure based on this idea is described in Figure 14, and it assumes that the model contains a data declaration

```
int isPostponed[t in Tasks] = 0;
```

to keep track of whether a task t is postponed. The search procedure illustrates many of the functionalities of OPL. First note the instructions

```

search {

forall(t in Tasks)
  onRange(a[t].start) do
    isPostponed[t] <- 0;

while not bound(a) do {
  select(t in Tasks : not bound(a[t].start) & not isPostponed[t]
    ordered by increasing dmin(a[t].start)) {
    let m = dmin(a[t].start) in {
      forall(o in Tasks)
        if isPostponed[o] & (dmax(a[o].start) <= m ∨ dmin(a[o].
          end) <= m) then
          fail
        endif;
      forall(t in Tasks:
        not isPostponed[t] & not bound(a[t].start) & dmin(a[t].
          start) = m
        ordered by increasing dmin(a[t].end))
        try
          a[t].start = m
          |
            isPostponed[t] <- 1
          endtry;
    };
  };
};
};
};

```

Fig. 14. A search procedure with dominance relations.

```

forall(t in Tasks)
  onRange(a[t].start) do
    isPostponed[t] <- 0;

```

which use the data-driven constructs of OPL and local assignments to make sure that a task t is not postponed when the range of its starting date changes. The data-driven constructs of OPL simplify the design of this search procedure by isolating this treatment from the time-directed labeling. The rest of the search procedure is a time-directed labeling enhanced to exploit dominance relations. The first enhancement is in the instructions

```

forall(o in Tasks)
  if isPostponed[o] & (dmax(a[o].start)<=m ∨ dmin(a[o].end)<=m) then
    fail
  endif;

```

that detect the dominance relation by considering all postponed activities and testing their latest starting time and earliest finishing date with respect to the selected time. The second enhancement is in the instructions

```
forall(t in Tasks:
  not isPostponed[t] & not bound(a[t].start) & dmin(a[t].start) = m
  ordered by increasing dmin(a[t].end))
try
  a[t].start = m
|
  isPostponed[t] <- 1
endtry;
```

that considers all activities that can be scheduled at time m . First note that only those activities that are not postponed are considered. In addition, observe that the second branch of the try instruction does not enforce the constraint $a[t].start > m$ but rather postpones the task until its range changes or a failure is detected.

7. RELATED WORK

As mentioned earlier, search procedures were integral parts of Constraint Logic Programming (CLP), the ancestor of modern constraint programming languages, since its inception and were in fact inherited from the “generate & test” or “test & generate” style of logic programming. For instance, all CLP programs solving combinatorial optimization problems (e.g., [Van Hentenryck 1989]) included search procedures. These were rather natural to express in CLP due to the nondeterministic nature of the languages. Indeed, CLP clauses directly describe a search tree that was explored by the depth-first strategy of CLP languages. Of course, other search strategies (e.g., LDS) can be programmed on top of a depth-first search as is commonly done in these languages nowadays.

Constraint libraries such as ILOG SOLVER [Puget 1994] were an important step in making constraint programming mainstream. Early versions of these libraries not only allowed for elegant, natural, and extensible constraint programs, but also made the computational model of CLP accessible from languages like C++. For instance, in ILOG SOLVER, search procedures are implemented in terms of objects called goals that can be combined with conjunctions and disjunctions to describe search trees. These trees were, once again, explored using depth-first search. Support for nested search was also provided.⁸ However, due to the deterministic nature of C++ and the nature of libraries, the search procedures are inherently less natural to read and write than in CLP languages.

The language ALMA-0 [Apt et al. 1998; Apt and Schaerf 1997] is an elegant language that introduces search abstractions from logic programming into an imperative language. Being a new language, it avoids the problem faced by constraint libraries and provides high-level and natural constructs to build

⁸Nested search can be easily implemented in CLP using a double negation as failure, an observation often attributed to Alain Colmerauer and part of the folklore of logic programming.

search trees. In fact, several of these constructs are closely related to those of OPL, although the two proposals were developed independently.⁹ In particular, ALMA-0 supports the equivalent of the `try` and `tryall` instructions, although no ordering is supported. ALMA-0 also supports other facilities to find all solutions (the counterpart in OPL is to `found` in `OPLSCRIPT`) and to commit to a solution.

The language 2LP [McAloon and Tretkoff 1995] is also an interesting and early language integrating search abstractions from logic programming and linear constraints over reals in a C-like language. Once again, it features the equivalent of the `try`, `tryall`, and `forall` instructions, although their syntax is not as rich as in OPL because of their underlying environment.

ELAN [Borovansky et al. 1998] is another innovative and interesting language, although it stems from fundamentally different motivations. ELAN is a nice realization of Kowalski's equation "algorithm = logic + control." The logic part of ELAN is a rewriting system, while the control part (called *strategy* in ELAN) is a separate language that specifies how to apply the rewriting rules. The control language offers primitives from nondeterminism, iteration, and some selection strategies (e.g., to collect only a subset of the results). The control program is specific to a set of rewriting rules (i.e., it is specified in terms of the rule names). Once again, the control language is not as rich as in OPL due to its rather different motivations. No specific support is provided in ELAN for exploration strategies at this point.

SALSA [Laburthe and Caseau 1998] is an algebraic search language that is also closely related to the search facilities of OPL, although, once again, the two languages were developed independently. For instance, SALSA supports, using a rather different syntax, the equivalent of the `tryall`, `select`, and `forall` instructions (with dynamic orderings) under the umbrella of a unique instruction. These instructions can be combined with various operators including sequencing, `commit`, nested search, and predefined incomplete search procedures. However, SALSA is a language to describe search trees (or partial search trees), and it has no specific support for implementing search strategies such as LDS and best-first search. One of the main novelties of OPL in this respect is to propose a single search language that makes it possible to compose, in a general fashion, search and strategy constructs.

Oz [Smolka 1995; Schulte 1997] is a concurrent constraint programming language that features computation states as first-class objects. Its main contribution as a constraint language is to have pioneered the ability to program search strategies in a generic way [Schulte 1997]. For instance, it is possible to write an Oz procedure implementing a generic LDS strategy or an Oz procedure implementing a generic best-first search. These procedures use the computation states, make a case analysis on the search tree, and often copy computation states to keep track of the various parts of the tree to be explored. In a sense, every such generic Oz procedure looks a bit like the OPL generic exploration algorithm shown in Figure 10. The main idea of OPL is to abstract these generic procedures further and to specify strategies using an evaluation and a postpone condition only. This provides, in our opinion, definitions of search procedures that are higher level and more adapted to modeling languages, since there is no need to manipulate the tree explicitly or to copy computation states. Of course,

⁹The first presentation of the main search features of OPL was in [Van Hentenryck 1997].

computation states have other applications besides exploration strategies and are an interesting concept per se [Schulte 2000].

Release 4.4 of ILOG SOLVER provides high-level abstractions for search strategies [Perron 1999]. The OPL support for strategies is heavily inspired by the ILOG SOLVER model, which also work in terms of the or-nodes of the search trees. An exploration strategy in ILOG SOLVER is defined in terms of a class that specifies how to evaluate a node and when to postpone a node. OPL mostly automates some of the routine aspects of these class definitions, letting users focus only the specification of the evaluation and postpone functions.

In summary, OPL provides, within a modeling language and at a very high level of abstraction, constructs that cover, to the best of our knowledge, the main search and strategy abstractions found in state-of-the-art constraint languages and libraries.

8. CONCLUSION

OPL is a modeling language for mathematical programming and combinatorial optimization problems. It is the first modeling language to combine high-level algebraic and set notations from modeling languages with a rich constraint language and the ability to specify search procedures and strategies that are the essence of constraint programming. This paper described the facilities available in OPL to specify search procedures. It describes the abstractions of OPL to specify both the search tree (search) and how to explore it (strategies). It also illustrated how to use these high-level constructs to implement traditional search procedures in constraint programming and scheduling.

ACKNOWLEDGMENTS

Many thanks to the three reviewers for their detailed comments and suggestions which helped us improve the presentation substantially.

REFERENCES

- APT, K., BRUNEKREEF, J., PARTINGTON, V., AND SCHAERF, A. 1998. Alma-O: An Imperative Language that Supports Declarative Programming. *ACM Transactions on Programming Languages and Systems* 20, 5 (September), 1014–1066.
- APT, K. AND SCHAERF, A. 1997. Search and Imperative Programming. In *Proc. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)* (January 1997), pp. 67–79. ACM Press.
- BISSCHOP, J. AND MEERAUS, A. 1982. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* 20, 1–29.
- BOROVANSKY, P., KIRCHNER, C., KIRCHNER, H., MOREAU, P., AND RINGEISSEN, C. 1998. An Overview of Elan. In *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications* (Pont-A-Mousson, France, September 1998).
- CARLIER, J. AND PINSON, E. 1990. A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. *Annals of Operations Research* 26, 269–287.
- CHENG, B., LEE, J., LEUNG, H., AND LEUNG, Y. 1996. Speeding up Constraint Propagation by Redundant Modeling. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)* (Cambridge, MA, August 1996). Springer Verlag.
- COLMERAUER, A. 1990. An Introduction to Prolog III. *Commun. ACM* 28, 4, 412–418.
- COLMERAUER, A. 1996. Spécification de Prolog IV. Technical report, Laboratoire d'informatique de Marseille.

- DEMEULESMEESTER, E. AND HERROELEN, W. 1992. A Branch and Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem. *Management Science* 38, 1803–1818.
- DINCBAS, M., SIMONIS, H., AND VAN HENTENRYCK, P. 1988. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Fifth International Conference on Logic Programming* (Seattle, WA, August 1988).
- DINCBAS, M., VAN HENTENRYCK, P., SIMONIS, H., AGGOUN, A., GRAF, T., AND BERTHIER, F. 1988. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, Japan, December 1988).
- FOURER, R., GAY, D., AND KERNIGHAN, B. 1993. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA.
- HARVEY, W. AND GINSBERG, M. 1995. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montreal, Canada, August 1995).
- ILOG OPL STUDIO 3.0. 2000. Reference Manual. Ilog SA, Gentilly, France.
- ILOG SOLVER 4.4. 1998. Reference Manual. Ilog SA, Gentilly, France.
- JAFFAR, J., MICHAYLOV, S., STUCKEY, P., AND YAP, R. 1992. The CLP(\mathcal{R}) Language and System. *ACM Trans. on Programming Languages and Systems* 14, 3, 339–395.
- LABURTHE, F. AND CASEAU, Y. 1998. SALSA: A Language for Search Algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)* (Pisa, Italy, October 1998).
- MAHER, M. 1987. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming* (Melbourne, Australia, May 1987), pp. 858–876.
- MARTIN, D. AND SHMOYS, P. 1996. A Time-based Approach to the Job-Shop Problem. In *Proc. of 5th International Conference on Integer Programming and Combinatorial Optimization (IPCO'96)* (Vancouver, Canada, 1996). Springer Verlag.
- MCAALON, K. AND TRETAKOFF, C. 1995. 2LP: Linear Programming and Logic Programming. In V. SARASWAT AND P. V. HENTENRYCK Eds., *Principles and Practice of Constraint Programming*. Cambridge, Ma: The MIT Press.
- PERRON, L. 1999. Search Procedures and Parallelism in Constraint Programming. In *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)* (Alexandria, VA, October 1999).
- PUGET, J-F. 1994. A C++ Implementation of CLP. In *Proceedings of SPICIS'94* (Singapore, November 1994).
- ROSEAUX. 1985. *Programmation linéaire et extensions; problèmes classiques*, Volume 3 of *Exercices et problèmes résolus de Recherche Opérationnelle*. Masson, Paris.
- SARASWAT, V. 1993. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA.
- SAVELBERGH, M. 1994. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA Journal of Computing* 6, 445–454.
- SCHULTE, C. 1997. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, Volume 1330 (Schloss Hagenberg, Linz, Austria, October 1997), pp. 519–533. Springer-Verlag.
- SCHULTE, C. 2000. Programming Deep Concurrent Constraint Combinators. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, Volume 1753 (Boston, MA, January 2000). Springer-Verlag.
- SMOLKA, G. 1995. The Oz Programming Model. In J. VAN LEEUWEN Ed., *Computer Science Today*. LNCS, No. 1000, Springer Verlag.
- VAN HENTENRYCK, P. 1989. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA.
- VAN HENTENRYCK, P. 1997. Visual Solver: A Modeling Language for Constraint Programming. In *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)* (Linz, Austria, October 1997). (Invited Talk).
- VAN HENTENRYCK, P. 1999. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass.
- VAN HENTENRYCK, P. AND DEVILLE, Y. 1991. 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).

VAN HENTENRYCK, P., MICHEL, L., LABORIE, P., NULJEN, W., AND ROGERIE, J. 1999. Combinatorial Optimization in OPL Studio. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence International Conference (EPIA'99)* (Evora, Portugal, September 1999). (Invited Paper).

VAN HENTENRYCK, P., MICHEL, L., PERRON, L., AND REGIN, J. 1999. Constraint Programming in OPL. In *Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP'99)* (Paris, France, September 1999). (Invited Paper).

Received November 1999; Revised February 2000; Accepted April 2000