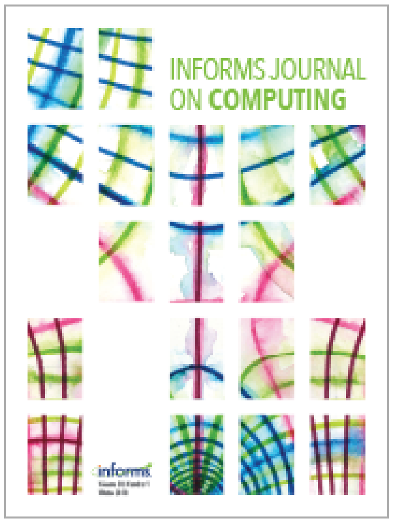

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Constraint and Integer Programming in OPL

Pascal Van Hentenryck,

Constraint and Integer Programming in OPL

Pascal Van Hentenryck
Department of Computer Science, Brown University, Box 1910,
Providence, Rhode Island 02912, USA
pvh@cs.brown.edu

In recent years, it has been increasingly recognized that constraint and integer programming have orthogonal and complementary strengths in stating and solving combinatorial optimization applications. In addition, their integration has become an active research topic. The optimization programming language OPL was a first attempt at integrating these technologies both at the language and at the solver levels. In particular, OPL is a modeling language integrating the rich language of constraint programming and the ability to specify search procedures at a high level of abstraction. Its implementation includes both constraint and mathematical programming solvers, as well as some cooperation schemes to make them collaborate on a given problem. The purpose of this paper is to illustrate, using OPL, the constraint-programming approach to combinatorial optimization and the complementary strengths of constraint and integer programming.
(*Artificial Intelligence; Computer Science; Integer Programming*)

1. Introduction

Combinatorial optimization problems are becoming increasingly important in a global and electronic economy. As is well-known, these problems are difficult, both from a computational complexity standpoint (they are NP-hard in general) and from a programming standpoint, since they require expertise in applied mathematics, algorithms, and software engineering.

In recent years, it has been increasingly recognized that integer programming and constraint programming have orthogonal and complementary strengths in approaching combinatorial applications. Informally speaking, integer programming focuses on the objective function by using a linear relaxation of the problem (possibly enhanced by problem-specific cuts) to prune suboptimal solutions. Constraint programming, in contrast, focuses on constraints using filtering algorithms to eliminate infeasible candidate solutions. It also became clear that some problems are best handled by integer programming, others by constraint programming, while some harder problems are currently outside the reach of both technologies. Simultaneously, it has been recognized that mathematical-modeling languages and constraint languages had complementary strengths. Modeling languages (e.g. Bisschop and Meeraus 1982, Fourer et al. 1993) provide high-level set and algebraic notations that may simplify the statement of the problem considerably, while constraint-programming languages feature rich constraint languages and the ability to specify search procedures at a high level. This recognition has led to increased research activity at the intersection of these two fields (e.g. Bockmayer and Kasper 1998, de Farias et al. 2001, El Sakkout and Wallace 2000, Hooker 2000, Hooker et al. 2001, Refalo 1999).

The OPL programming language (Van Hentenryck 1999, Van Hentenryck et al. 2000) was a first attempt to support this complementarity at the language and solver levels. From a language standpoint, OPL is a modeling language supporting the traditional set and algebraic notations of mathematical languages, while integrating the rich language of constraint programming, including the ability to specify search procedures. From a constraint-solving standpoint, the OPL implementation includes both constraint and mathematical programming solvers as well as some cooperation schemes to benefit from both technologies. This paper describes some of the features of OPL in this area. Its main purpose is threefold:

1. It gives an overview of the language of constraint programming, i.e., its variety of constraints and its high-level constructs to specify search procedures;
2. It gives an overview of the computational model of constraint programming, i.e., how constraint programming approaches the solving of combinatorial optimization problems;
3. It illustrates several cooperation schemes between constraint and integer programming.

It is impossible to do justice to such a broad topic in a single paper. The approach chosen in this paper is to be concrete, to select a small number of case studies to illustrate these features, and to complement them with some more conceptual sections. In so doing, we leave out many interesting features of constraint programming and OPL, as well as many promising cooperation schemes. However, the hope is that the paper will stimulate readers to follow the references in order to learn more about these topics.

It is also important to mention that OPL is part of a larger system that also includes OPLScript (Van Hentenryck and Michel 2000), the OPL component library, and a development environment. OPLScript is a script language where models are first-class objects and that is useful to approach applications requiring the solving of a sequence of models, multiple instances of the same model, or decomposition schemes such as column generation and Benders decomposition algorithms. The OPL component library makes it possible to integrate an OPL model directly in an application by generating C++ code or by using a COM component. These features of the OPL system are not discussed in the paper.

The rest of this paper is organized as follows. Section 2 introduces constraint programming through a simple example. Section 3 gives an overview of constraint programming in a more conceptual way. Section 4 describes a constraint-programming solution to a sport-scheduling application. Section 5 presents some cooperation schemes for constraint and integer programming. Sections 6 and 7 illustrate the two cooperation schemes. Section 8 concludes the paper.

2. Getting Started

This section considers a simple example, the *n*-queens problem, and describes some constraint-programming models to solve it. Its goal is to illustrate the computational model of constraint programming on a simple example, to introduce some basic elements of OPL, to describe basic principles underlying constraint-programming search, and to present the idea of redundant modeling, which is the basis for the cooperation scheme presented later in the paper.

2.1. A Simple Model

The *n*-queens problem consists of placing *n* queens on a chessboard of size $n \times n$ so that no two queens lie on the same row, column, or diagonal. As is typical in constraint (and mathematical) programming, the first step in solving a problem consists of choosing the variables, often called *decision variables*, in terms of which the constraints are then stated. Since no two queens can be placed on the same column, a simple model of this problem consists of associating a queen with each column and searching for an assignment of rows to the queens so that no two queens are placed on the same row or on the same diagonals. Figure 1 depicts an OPL statement for the *n*-queens problem.

```
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;
   }
};
```

Figure 1 **A Simple N-Queens Model**

The statement illustrates portions of the structure typically found in constraint programs: the declaration of the data, the declaration of decision variables, and the statement of constraints. More involved applications often include a search procedure, as will be illustrated shortly. The model first declares an integer `n`, whose initialization is given in a separate file, and a range `Domain`. It then declares an array of n variables, all of which take their values in the range `1...n`. In particular, variable `queen[i]` represents the row assigned to the queen placed in column `i`. The `solve` instruction defines the problem constraints, i.e., that no two queens should attack each other. It indicates that the purpose of this model is to find an assignment of values to the decision variables that satisfies all constraints. The basic idea in this model is to generate, for all $1 \le i < j \le n$, the constraints

```
queen[i] <> queen[j]
queen[i] + i <> queen[j] + j
queen[i] - i <> queen[j] - j
```

where the symbol <> means "not-equal." OPL uses a single universal quantifier for introducing parameters `i` and `j` and specifies that these parameters are ordered, i.e., `i < j`. Note that there is no objective function in this model, since we are only interested in finding an assignment of values to the variables that satisfies all constraints.

One of the fundamental ideas of constraint programming is to associate a domain with each decision variable and to use constraints to reduce these domains (Van Hentenryck and Dincbas 1986). The domain-reduction process, often called constraint propagation or domain filtering, prunes the search space by removing, from the domains, values that cannot appear in any solution. When no more domain reductions are possible, a constraint-programming system typically makes a choice, e.g., it assigns a value to a variable. As a consequence, the computational model of constraint programming systems can be viewed, in a first approximation, as the iteration of two steps: a constraint propagation step that reduces the variable domains and a decomposition step that partitions the problem into several simpler subproblems that are solved independently. Note that a problem or a subproblem does not contain any solution when the domain of a variable becomes empty.

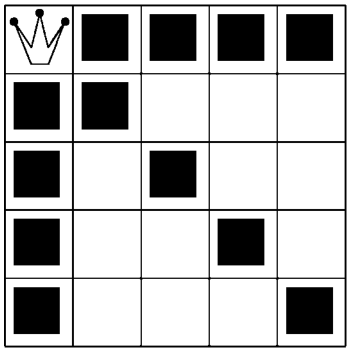
Figure 2 The Five-Queens Problem After One Choice

To illustrate the computational model, consider first the five-queens problem. Constraint propagation does not reduce the domains of the variables initially. OPL thus generates a value, say 1, for one of its variables, say `queen[1]`. After this nondeterministic assignment, constraint propagation removes inconsistent values from the domains of `queen[2]`,..., `queen[5]`, as depicted in Figure 2. The next step of the generation process tries the value 3 for `queen[2]`. OPL then removes inconsistent values from the domains of the remaining queens (see Figure 3). Since only one value remains for `queen[3]` and `queen[4]`, these values are immediately assigned by OPL to these variables and, after more constraint propagation, OPL assigns the value 4 to `queen[5]`. A solution to the five-queens problem is thus found with two choices and without backtracking, i.e., without reconsidering any of the choices made by OPL.

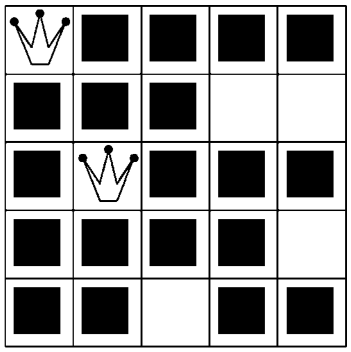
Figure 3 The Five-Queens Problem After Two Choices

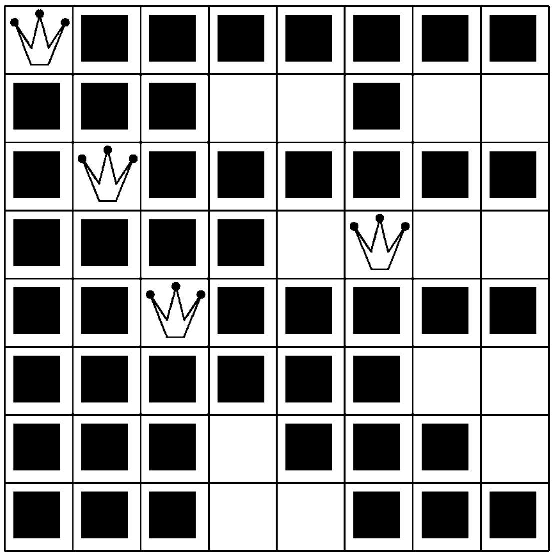

Figure 4 The Eight-Queens Problem After Three Choices (Intermediate)

It is also interesting to consider the eight-queens problem after three choices. Figure 4 depicts an intermediate stage when some (but not all) constraint propagation has taken place. Since only one value is left for queen[6], OPL assigns this value and propagates the constraints further. This assigns the value 7 to queen[8] which, in turn, assigns the value 2 to

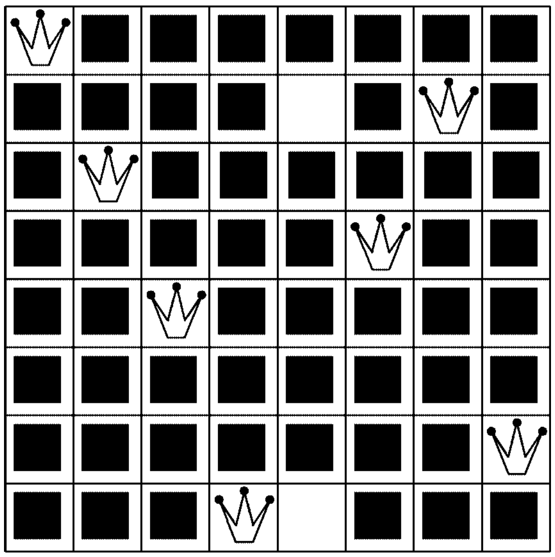

Figure 5 The Eight-Queens Problem After Three Choices (Intermediate)

queen[7] and the value 8 to queen[4] (see Figure 5). As a consequence, queen[5] has no value left in its domain and constraint solving fails. OPL then backtracks and tries another value for queen[3].

Observe that the constraint-propagation step and the search procedure are extremely simple in the queens problem but they may become rather complex in some advanced constraint-programming applications, as discussed later in the paper.

2.2. Search Procedure

This section shows how to add a search procedure to the above model. Its main purpose is to illustrate this important functionality of constraint programming and to introduce some basic search constructs of OPL that are used later in the paper.

Specifying a search procedure is often desirable on practical applications to exploit the structure of the problem during branching. Constraint programming, and OPL in particular, offers a clean separation between the constraint statements and the search procedure description. Consider Figure 6 that describes a model for the queens problem with the search procedure

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

```
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;
};
```

Figure 6 A Search Procedure for the *n*-Queens Model

The basic idea behind this search procedure, often called a *standard labeling routine*, is to consider each decision variable in sequence and to generate, in a nondeterministic way, a value for each of them. If, at some computation stage, a failure occurs (i.e., the domain of a variable becomes empty), OPL backtracks and tries another value for the last queen assigned. More precisely, the `forall` instruction corresponds to an ordered *and-node* in artificial intelligence terminology and executes its body for each value `i` in `Domain`. The `tryall` instruction corresponds to an ordered *or-node* in artificial intelligence terminology and is nondeterministic, i.e., it specifies a choice point with a number of alternatives and one of them must be selected. In the context of the queens problem, each of these alternatives corresponds to the assignment of a value to a queen. Consider the instruction

```
tryall(v in Domain)
  queen[1] = v;
```

This instruction first tries to assign value 1 to `queen[1]`. If this choice leads to a failure (either during constraint propagation or later in the search), the `tryall` instruction tries the next available value or fails if none are left. It is important to stress a few points at this stage.

• First, OPL clearly separates the constraint specification and the search procedure. However, although these two components are *physically separated* in the problem statement, they are *cooperating* in solving the problem as was demonstrated earlier. Indeed, each time the search procedure makes a choice (e.g., each time it assigns a value to a variable), it initiates a constraint-propagation step that reduces the domains. Moreover, if the constraint propagation fails by making the domain of a variable empty, the search procedure is informed: it must abandon the current computation and reconsider one of its choices.

• Second, the search procedure specifies the *search tree* to explore but it does not specify *how* to explore it. By default, OPL, and most constraint-programming systems, uses depth-first search to explore the search tree. However, modern constraint-programming languages such as OPL support many other exploration strategies and even let users specify their own.

2.3. Constraint-Programming Heuristics

In integer programming, it is common to use the solution of the linear relaxation to guide the search, e.g., by choosing which variable to branch on. Constraint programming often uses complementary information from the domain store and/or the contraints (e.g., the tightness of a constraint). This section briefly illustrates this idea.

The main step of the standard labeling procedure consists of choosing a variable to assign and to choose a value for this variable in a nondeterministic way. The choice of which variable to assign next is often important in practice and can produce substantial differences in execution time. A general principle to guide the choice of which variable to assign next is the so-called *first-fail* principle. This heuristic recommends to try first where one is most likely to fail. The first-fail principle is often effective in reducing the size of the search space by discovering failures early and by producing smaller search trees. (The first-fail principle is not always the most effective branching heuristic. As mentioned by one of the reviewers of this paper, it assumes a certain connectivity between the variables. In addition, there are applications where the exact opposite heuritic, i.e., selecting the variable with the largest domain, is more effective. This is especially the case in nonlinear constraint systems over the reals (Van Hentenryck et al. 1997)). When applied to the variable choice in the queens problem, this principle may be interpreted as choosing the variable that has the smallest domain, since it seems that this variable will be harder to assign. Note also that choosing the variable with the smallest domain leads, in general, to smaller search trees, since fewer alternatives need to be considered. Figure 7 depicts a model for the queen problem with a dynamic variable ordering that selects the variable with the smallest domain first. The only difference is the search procedure

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

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

Figure 7 An *n*-Queens Model with a Dynamic Variable Ordering

that now specifies an ordering for the variable choice. Intuitively, the model behaves as follows. After an initial constraint-propagation step, the search procedure selects the variable that has the smallest domain as specified by the words `ordered by increasing dsize(queen[i])`, where `dsize(queen[i])` returns the number of values in the domain of `queen[i]`. It generates a value for this selected variable and propagates the constraints, which reduces the variable domains. The search procedure then selects the next variable to assign by considering these reduced domains, generates a value for this newly selected variable, and iterates this process until all variables are instantiated. It is important to stress that the variable ordering is *dynamic*, i.e., the next variable to assign is chosen with respect to the current domains. The variable choice can also be further refined. Consider 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;
};
```

which uses a lexicographic criterion for the variable choice. It chooses the variable with the smallest domain and, in case of ties, the variable closest to the middle of the chessboard. This type of lexicographic variable ordering is often used in constraint programming.

The value choice can also be specified in a dynamic fashion in the `tryall` instruction. Once again, it is typical to use information from the constraint store to guide this choice. This is illustrated in the next section, once the idea of *redundant modeling* is introduced.

2.4. Redundant Modeling

The cooperation schemes presented later in the paper are based on the idea of *redundant modeling*. It consists of using different models of the same application and making them cooperate through constraints. See Cheng et al. (1996) and van Emden (1999) for discussions of the benefits of redundant modeling.

Figure 8 illustrates redundant modeling on the *n*-queens problem. It models the problem using both a column and a row viewpoint, using constraints to link them together. Recall that the earlier models associate, with each column, a variable whose value denotes the row of the queen placed on this column. This is, of course, an arbitrary decision. We could have chosen to associate a variable with every row. The variable would then denote the column of the queen on this row. The redundant model uses these two formulations and makes them cooperate through constraints.

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

```
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,v]))
         queen[col,i] = v;
};
```

Figure 8 Redundant Modeling for the *n*-Queens Program

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. The link between the two formulations is achieved by constraints of the form

```
queen[col, i] = v <=> queen[row, v] = i;
```

Informally speaking, this constraint specifies that, if the queen placed on column `i` is in row `v`, then the queen on row `v` must be placed on column `i` and vice versa. These constraints also illustrate that constraint programming makes it possible to use logical combinations of constraints.

The search procedure is particularly interesting in this model. It uses the same variable ordering as before but it now adds a dynamic value ordering, i.e., it specifies the order in which values are tried. The 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. Many search procedures in constraint programming, although more involved in general, are based on similar principles.

2.5. Comparison with an Integer-Programming Model

We now present an integer-programming model for the queens problem. *The model illustrates a fundamental difference between constraint and integer programming models: the choice of the problem variables.* (Section 6 also illustrates this difference on a scene allocation problem.) As discussed earlier, the first step in solving a combinatorial problem with constraint or integer programming consists of choosing the problem variables in terms of which constraints are then stated. In general, variables in constraint programming directly model the "natural" decision variables of the problem. For instance, in the queens problem, a variable denotes the position of the queen to be placed on a given column. Integer programming, on the other hand, often use of more "primitive" 0/1 variables representing simpler binary decisions. In both cases, the choice of the decision variables is motivated by the underlying solver. Binary variables in integer programming make it easier to build strong linear formulations, while more "global" variables in constraint programming enable the formulation to capture combinatorial substructures more easily.

Figure 9 depicts an integer-programming model for the queens problem in OPL. Observe the use of variables of the form queen[i,j], which is 1 if the queen on row i is placed on column j, and 0 otherwise. This choice of variables makes it easy to state linear constraints to model the row, the column, and the diagonal constraints. The first set of constraints

```
forall(i in 1..n)
  sum(j in 1..n) queen[i, j] = 1;
```

expresses that exactly one queen must be placed on each row. The second set of constraints expresses that exactly one queen must be placed on each column

```
forall(i in 1..n)
  sum(j in 1..n) queen[j,i] = 1;
```

Observe that this set of constraints is implicit in the standard constraint-programming statement. The third (resp. fourth) set of inequalities ensures that at most one queen is placed on downwards (resp. upwards) diagonals. As is often the case, the integer-programming statement is less natural than the constraint-programming model. (Observe that it is possible to design an integer-programming model using variables as in the contraint statement by using big M transformations of disequations, e.g., Van Hentenryck 1989. However, big M transformations generally introduce weaker integer programming formulations.)

```
int n = ...;
var int queen[1..n,1..n] in 0..1;
solve {
   forall(i in 1..n)
      sum(j in 1..n) queen[i,j] = 1;
   forall(i in 1..n)
      sum(j in 1..n) queen[j,i] = 1;
   forall(j in 0..n-1) {
      sum(k in 1..n-j) queen[k,k+j] <= 1;
      sum(k in 1..n-j) queen[k+j,k] <= 1;
   };
   forall(j in 0..n-1) {
      sum(k in 1..n-j) queen[n+1-k,k+j] <= 1;
      sum(k in 1..n-j) queen[n+1-k-j,k] <= 1;
   };
};
```

Figure 9 An Integer-Programming Model for the Queens Problem

3. A Brief Overview of Constraint Programming

This section gives a brief overview of constraint programming. Once again, the purpose is not to be exhaustive (see, for instance, overviews of constraint programming, or subfields of constraint programming: Jaffar and Maher 1994, Marriott and Stuckey 1999, Saraswat 1989, Van Hentenryck 1991, 1997, Van Hentenryck and Saraswat 1996) but to survey what we consider to be some of the fundamental ideas in this field.

3.1. What Is Constraint Programming?

Constraint programming is a recent entry to the field of programming languages. Its essence is a two-level architecture integrating a *constraint* and a *programming* component. The constraint component provides the basic operations of the architecture and consists of a system of reasoning about fundamental properties of constraint systems such as satisfiability and entailment. The constraint component is often called the *constraint store*, by analogy to the memory store of traditional programming languages. The constraint store contains the constraints accumulated at some computation step and supports various queries and operations over these constraints. Operating around the constraint store is a programming-language component that specifies how to combine the basic operations, often in nondeterministic ways, since search is so fundamental in many applications.

The constraint-programming framework has been applied to many areas, including computer graphics (e.g., Borning 1981), software engineering (e.g., test generation in Gotlieb et al. 2000), databases (e.g., Kanellakis et al. 1990), hybrid systems (e.g., Kohn et al. 1995), finance (e.g., Berthier 1988, Huynh and Lassez 1988), engineering (e.g., Graf et al. 1989, Heintze et al. 1987), circuit design (e.g., Simonis and

Dincbas 1987), and, of course, combinatorial optimization. Given the diversity of these application areas, it is not surprising that the programming and constraint components can be of fundamentally different nature. However, when restricting attention to combinatorial optimization, constraint programming systems are generally based on a common set of design principles that stem from their roots in Constraint Logic Programming (CLP) (Colmerauer 1982, 1990; Jaffar and Lassez 1987; Van Hentenryck 1987b). More precisely, the constraint programming approach to combinatorial optimization can be characterized (at this point) by three main features:

1. an expressive language to state combinatorial-optimization problems, including a rich constraint language and the ability to specify search procedures;
2. a new computational model to solve combinatorial-optimization problems that works on discrete substructures of the application;
3. a flexible architecture that makes it natural to support cooperative solvers.

This section reviews these three features independently and restricts attention to combinatorial optimization only. We conclude the section with a brief comparison to integer programming.

3.2. The Language of Constraint Programming

As shown in Section 2, solving a combinatorial-optimization problem in constraint programming amounts to

1. describing the problem constraints;
2. specifying a search procedure.

In OPL, the second step is not strictly necessary but, in general, it is important to provide a search procedure to achieve a reasonable efficiency.

Constraints in constraint programming are generally expressed in a rich language that includes linear and nonlinear constraints, the ability to index arrays with variables (the so-called ELEMENT constraint of CHIP (Van Hentenryck 1987b, 1989; Van Hentenryck and Carilon 1988)), logical combinations of constraints, cardinality constraints, and higher-order constraints, all pioneered by cc(FD) (Van Hentenryck et al. 1995), as well as global constraints. Global constraints were present in CHIP as early as 1987; see the ELEMENT constraint in Van Hentenryck (1987b) and the ATMOST constraint in Dincbas et al. (1988). However, they became a fundamental research topic only in the 1990s when they were found to provide a natural way to integrate many algorithms from theoretical computer science, operations research, and numerical analysis (e.g., Beldiceanu and Contejean 1994, Caseau and Laburthe 1997, Nuijten 1994, Regin 1994, Van Hentenryck 1997). A global constraint captures a substructure that arises in many applications and is amenable to efficient pruning algorithms (as described later in this section). In addition, some constraint languages also offer set variables and set constraints (e.g., Gervet 1994). Many of these constraints are illustrated in this paper.

The ability to specify a search procedure is another fundamental feature of constraint-programming languages. Once again, this functionality was present in the early CLP languages and was considered fundamental to obtain reasonable efficiency on difficult combinatorial problems (e.g., Van Hentenryck 1989). As mentioned earlier, much research in constraint programming was devoted to finding ways of expressing search procedures at a high level. Concurrent constraint programming (Maher 1987, Saraswat 1989), which is the foundation of constraint languages such as Oz (Smolka 1995) and cc(FD) (Van Hentenryck et al. 1995), introduced a constraint-driven computation where the programming component is a set of agents communicating by adding constraints to, and querying, the constraint store. High-level constructs such as `forall` and `tryall` were pioneered by SALSA (Laburthe and Caseau 1998) and OPL (Van Hentenryck et al. 2000). The language Oz (Schulte 1997) pioneered novel features to express search strategies (e.g., limited discrepancy search Harvey and Ginsberg 1995) in consice ways. Support for search strategies are now available in several modern languages (Perron 1999, Van Hentenryck et al. 2000). Finally, hybrid algorithms, combining systematic and local search, were also investigated in (Laburthe and Caseau 1998, Pesant et al. 1997, Shaw 1998).

It is also important to stress that the constraint-programming framework is essentially language-independent. Early constraint languages were based on logic programming (Colmerauer et al. 1973, Kowalski 1974). The embedding of constraints in more traditional languages such as C and C++ was pioneered by CHARME which was based on the CHIP implementation. CHARME was in fact commercialized by BULL in the late 1980s. Other examples of libraries and languages integrating constraints in procedural or object-oriented languages include 2LP(McAloon and Tretkoff 1995), CLAIRE (Laburthe and Caseau 1998), and the library (Puget and Leconte 1995). Note however that declarative, relational, and/or nondeterministic languages provide a more natural vehicle to embed constraint programming because of their conceptual commonalities. One of the design goals of OPL was to reconcile the elegance of search in declarative languages with the high-level data structures typically found in modeling and object-oriented languages.

3.3. The Computation Model of Constraint Programming

Section 2 illustrated the computational model of constraint programming for combinatorial optimization. In particular, it indicated that

1. The computation model of constraint programming is based on a tree-search algorithm that applies a constraint-satisfaction algorithm at each node;
2. During constraint satisfaction, constraints are processed *independently* (or *locally*) and interact only by reducing the domains of the variables and by adding constraints.

The first element of the computational model is, in essence, similar to the branch-and-bound approach to integer programming. The main originality in constraint programming lies in the underlying constraint-satisfaction algorithms or, more generally, in the constraint-store organization.

Figure 10 depicts the architecture of the constraint store. The core of the architecture is the *domain store* that associates a domain with each variable. The domain of a variable represents its set of possible values at a given computation state (e.g., at a node in the tree search). Gravitating around these domains are the various constraints, each of which has no knowledge of the other constraints. Associated with each constraint is a constraint-satisfaction algorithm whose

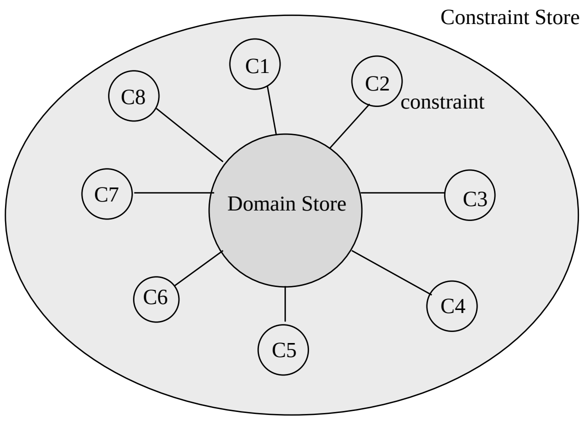

Figure 10 The Constraint Store of Constraint Programming

primary role is to perform two main tasks:

1. to determine if the constraint is consistent with the domain store, i.e., if there exist values for the variables in their domains that satisfy the constraint;
2. to apply a filtering algorithm to remove, from the domains of the constraint variables, values that do not appear in any of its solutions.

In addition, the constraint-satisfaction algorithm may add new constraints to the constraint store as we discuss later in the paper.

The constraint solver can then be viewed as a simple iterative algorithm whose basic step consists of selecting a constraint and applying its constraint-satisfaction algorithm. The algorithm terminates when a constraint is inconsistent with respect to the domain store or when no more domain reductions are possible. Note that, on termination, there is no guarantee that there exists a solution to the constraint store. The constraints may all have a local solution with respect to the domain store but these solutions may be incompatible globally. This architecture, which is now the cornerstone of most modern constraint-programming systems, was pioneered by the CHIP system (Dincbas et al. 1988, Van Hentenryck 1987b, Van Hentenryck 1989, Van Hentenryck and Dincbas 1986). CHIP included a solver for discrete finite domains based on constraint-satisfaction techniques (Waltz 1972, Montanari 1974, Mackworth 1977).

It is important to emphasize that, unlike integer and linear programming, constraint-programming systems may support *arbitrarily complex constraints*. These constraints are not restricted to linear constraints, or even nonlinear constraints, and they may represent complex relations between their variables. For instance, a constraint may require that all its variables be assigned distinct values or that a set of activities do not overlap in time. As a consequence, it is useful, and correct, to think of a constraint as representing *an interesting subproblem* of the application and it is one of the fundamental issues in constraint programming to isolate classes of constraints that are widely applicable and amenable to efficient implementation.

It is also important to stress that, ideally, the constraint-satisfaction algorithm associated with a constraint should be complete (i.e., it should remove all inconsistent values) and run in polynomial time. Such complete algorithms enforce arc consistency, a concept originating from the artificial intelligence community (Mackworth 1977, Waltz 1972). However, enforcing arc consistency may sometimes prove too hard (i.e., it may be an NP-hard problem), in which case simpler consistency notions are defined and implemented. This is the case, for instance, in scheduling algorithms. (More complex consistency notions also exist: e.g., path consistency; Montanari 1974 and are supported by some constraint languages. They have been instrumental in solving some hard nonlinear problems Van Hentenryck 1997). The constraint satisfaction algorithm should also be incremental, since it is executed at each node of the search tree (sometimes several times) with domain stores that are closely related in general.

3.4. Cooperation in Constraint Programming

Cooperating solvers have always been an active research area in constraint programming. PROLOG III (Colmerauer 1990) was the pioneering system in that respect. It included a linear-programming solver, a Boolean solver, as well as a constraint solver for equations over lists. These solvers interacted either explicitly by posting constraints or through variable bindings. Further research led to systems such as CLP(BNR) (Benhamou and Older 1997), Prolog IV (Colmerauer 1996), and ICE (Beringer and De Backer 1994), which anticipated some of the recent activities on cooperation (e.g., El Sakkout and Wallace 2000, Hooker et al. 2001, Refalo 1999)). Observe also that linear-programming solvers were part of constraint-programming systems since their early days.

3.5. Comparison with Integer Programming

It is interesting to contrast constraint and integer programming (e.g., Wolsey 1998) and show why they appear complementary for combinatorial optimization. (We only discuss the branch-and-bound approach to integer programming and ignore several other important areas such as Lagrangian relaxation and local search techniques.) The two approaches have commonalities and specificities. Both are based on systematic search and tackle issues such as how to prune the search tree and how to search the search space effectively. They differ in how they address these issues.

In integer programming, the focus is on the objective function and pruning eliminates suboptimal solutions by computing a lower bound (in the case of minimization problems) at every node of the search tree. The lower bound is obtained by solving a linear relaxation of the problem, i.e., by expressing the problem as a set of linear constraints, relaxing the integrality constraints, and using a linear-programming algorithm. The linear relaxation is generally strengtened by adding problem-specific cuts. Integer-programming systems also use information from the linear-programming relaxation to search more effectively, e.g., to choose which variable to branch on.

In constraint programming, the focus is on the problem constraints and the pruning of infeasible candidate solutions. The pruning is achieved by applying specialized filtering algorithms that reduce the domain store and detect inconsistencies. These filtering algorithms handle discrete substructures of the problem directly (through global constraints) but constraints are considered only locally, i.e., distinct constraints only interact through the domain store. Constraint-programming systems also use information from the domain store (e.g., the number of feasible values for a variable) and from the constraints (e.g., the tightness of a constraint) to guide the search process.

Of course, this comparison exaggerates the difference between the two approaches. When advanced techniques such as probing (Savelbergh 1994) or sophisticated global constraints (e.g., viewing linear programs as global constraints) are taken into account, the differences between constraint and integer programming systems start to blur. But the comparison still captures, we believe, a fundamental methodological difference between the two approaches. It also highlights the fact that the two approaches seem orthogonal and complementary and it seems natural to try exploiting the strengths of the two approaches, relying on integer programming for optimizing and on constraint programming for exploring the space of feasible solutions.

4. Sport Scheduling

This section describes constraint programming solutions to a sport-scheduling problem that is a well-known benchmark in mathematical programming (problem 10TEAMS from the MIPLIB library). The problem is well-suited for constraint programming, since it is purely combinatorial. It also illustrates several interesting features of constraint programming, including some well-known global constraints, symbolic constraints, and the important role of arc consistency. Section 4.1 describes the problem. Section 4.2 presents an OPL model that solves the 14-team problem in about 44 seconds. Section 4.3 show how to specialize it further to find a solution for 14 to 30 teams quickly. Both models are based on the constraint programs presented in Régin (1998) (See also Van Hentenryck et al. 1999).

4.1. Problem Description

The sport-scheduling problem (McAloon et al. 1997, Régin 1998) is a standard benchmark submitted by Bob Daniel to the well-known MIP library. It consists of scheduling games between n teams over $n-1$ weeks. In addition, each week is divided into $n/2$ periods. The goal is to schedule a game for each period of every week so that the following constraints are satisfied:

1. Every team plays against every other team;
2. A team plays exactly once a week;
3. A team plays at most twice in the same period over the course of the season.

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
period 1	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

Figure 11 A Solution to the Sport-Scheduling Problem with Eight Teams

A solution to this problem for 8 teams is shown in Figure 11. In fact, the problem can be made more uniform by adding a "dummy" final week and requesting that all teams play exactly twice in each period. The rest of this section considers this equivalent problem for simplicity. Note also that it is claimed in McAloon et al. (1997) that state-of-the-art MIP solvers cannot find a solution for 14 teams. Some early constraint and local-search models were also presented in McAloon et al. (1997).

4.2. A Simple OPL Model

The simple model is depicted in Figure 12. Its input is the number of teams `nbTeams`. Several ranges are defined from the input: the teams `Teams`, the weeks `Weeks`, and the extended weeks `EWeeks`, i.e., the weeks plus the dummy week. The model also declares an enumerated type `slot` to specify the team position in a game (`home` or `away`). The declarations

```
int occur[t in Teams] = 2;
int values[t in Teams] = t;
```

specifies two arrays that are initialized generically and are used to state constraints later on. The array `occur` can be viewed as a constant function always returning 2, while the array `values` can be tought of as the identify function over teams.

The main modeling idea in this model is to use two classes of variables: team variables that specify the team playing on a given week, period, and slot, and the game variables specifying which game is played on a given week and period. The use of game variables makes it simple to state the constraint that every team must play against each other team. Games are uniquely identified by their two teams. More precisely, a game consisting of home team `h`

```
int nbTeams = ...;
range Teams 1..nbTeams;
range Weeks 1..nbTeams-1;
range EWeeks 1..nbTeams;
range Periods 1..nbTeams/2;
range Games 1..nbTeams*nbTeams;
enum Slots = { home, away };

int occur[t in Teams] = 2;
int values[t in Teams] = t;

var Teams team[Periods,EWeeks,Slots];
var Games game[Periods,Weeks];

struct Play { int h; int a; int g; };
{Play} Plays = { <i,j,(i-1)*nbTeams+j> | ordered i, j in Teams };
predicate link(int h,int a,int g) in Plays;

solve {
   forall(w in EWeeks)
      alldifferent( all(p in Periods & s in Slots) team[p,w,s]);
   alldifferent(game);
   forall(p in Periods)
      cardinality(occur,values,all(w in EWeeks & s in Slots) team[p,w,s]);
   forall(p in Periods & w in Weeks)
      link(team[p,w,home],team[p,w,away],game[p,w]);
};
search {
   generate(game);
};
```

Figure 12 A Simple Model for the Sport-Scheduling Problem

and away team a is uniquely identified by the integer (h-1)∗nbTeams + a. The instruction

```
var Teams team[Periods, EWeeks, Slots];
var Games game[Periods, Weeks];
```

declares the variables. These two sets of variables must be linked together to make sure that the game and team variables for a given period and a given week are consistent. The instructions

```
struct Play { int h; int a; int g; };
{Play} Plays = { <i,j,(i−1)∗nbTeams
               +j> | ordered i, j in Teams };
```

specify the set of legal games Plays for this application. For 8 teams, this set consists of tuples of the form

```
<1,2,2>
<1,3,3>
...
<7,8,56>
```

Note that this definition eliminates some symmetries in the problem statement since the home team is always smaller than the away team. The instruction

```
predicate link(int h,int a,int g) in Plays;
```

defines a symbolic constraint by specifying its set of tuples. In other words, link(h,a,g) holds if the tuple <h,a,g> is in the set Plays of legal games. This symbolic constraint is used in the constraint statement to enforce the relation between the game and the team variables. The constraint declarations follow almost directly the problem description. The constraint

```
forall(w in EWeeks)
     alldifferent(all(p in Periods & s in Slots)
                  team[p,w,s]);
```

specifies that all the teams scheduled to play on week `w` must be different. It uses an aggregate operator `all` to collect the appropriate team variables by iterating over the periods and the slots. The `alldifferent` constraint is probably the most well-known global constraint and it captures an important substructure of the application. The OPL implementation enforces arc consistency on this constraint.

Definition 1. *Arc Consistency.* A constraint $C(x_1, \ldots, x_n)$ is arc-consistent in domains $(D_1, \ldots, D_n)$ if, for each variable x_i and each value v_i in D_i, there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in domains $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $C(v_1, \ldots, v_n)$ holds. An arc-consistency algorithm for constraint C and domains $(D_1, \ldots, D_n)$ returns the largest domains $(D_1^*, \ldots, D_n^*)$ such that C is arc-consistent in $(D_1^*, \ldots, D_n^*)$.

Arc consistency on an `alldifferent` constraint can be enforced by applying a matching algorithm on a bipartite graph (where one of the set represents the variables and the other set represents their possible values) together with a search for strongly connected components on the same graph (Costa 1994, Régin 1994). Consider now the constraint

```
cardinality(occur,values,all(w in EWeeks &
             s in Slots) team[p,w,s]);
```

specifies that a team plays exactly twice over the course of the "extended" season. Its first argument specifies the number of occurrences of the values specified by the second argument in the set of variables specified by the third argument that collects all variables playing in period `p`. Cardinality is another well-known global constraint. Once again, the OPL implementation achieves arc consistency on this constraint that, in this case, involves the solving of a feasible flow and, once again, a search for connected components in the residual graph (Régin 1996). The constraint

```
alldifferent(game);
```

specifies that all games are different, i.e., that each team plays against every other team. Finally, the constraint

```
link(team[p,w,home],team[p,w,away],game[p,w]);
```

is most interesting. It specifies that the game `game[p,w]` consists of the teams `team[p,w,home]` and `team[p,w,away]`. OPL enforces arc consistency on this symbolic constraint as well. (Observe that this constraint is equivalent to `(team[p,w, home],team[p,w,away],game[p,w]) in Plays;`. This syntax was not available in OPL howver.)

The search procedure in this statement is extremely simple and consists of generating values for the games using the first-fail principle. Note also that generating values for the games automatically assigns values to the team by constraint propagation.

As mentioned, this model finds a solution for 14 teams in about 44 seconds on a 400 mhz PC. It is also interesting to stress that all constraints in this problem, i.e., all the `alldifferent`, `cardinality`, and symbolic constraints are handled independently and only interact through the domain store. It is also critical to enforce arc consistency to obtain a reasonable efficiency: only reducing the bounds would lead to a substantial slowdown in efficiency. (As indicated by one of the reviewers, bound reduction has no real motivation here since the domains do not represent numbers.)

4.3. A Round-Robin Model

The simple model has many symmetries that enlarge the search space considerably. In this section, we describe a model that uses a round-robin schedule to determine which games are played in a given week. As a consequence, once the round-robin schedule is selected, it is only necessary to determine the period of each game, not its schedule week. In addition, it turns out that a simple round-robin schedule makes it possible to find solutions for large numbers of teams. The model is depicted in Figures 13 and 14.

The main novelty in the statement is the array `roundRobin` that specifies the games for every week.

```
int nbTeams = ...;
range Teams 1..nbTeams;
range Weeks 1..nbTeams-1;
range EWeeks 1..nbTeams;
range Periods 1..nbTeams/2;
range Games 1..nbTeams*nbTeams;
enum Slots = { home, away };
int occur[t in Teams] = 2;
int values[t in Teams] = t;
struct Play { int f; int s; int g; };
{Play} Plays = { <i,j,(i-1)*nbTeams+j> | ordered i, j in Teams };
predicate link(int f,int s,int g) in Plays;
Play roundRobin[Weeks,Periods];
initialize {
   roundRobin[1,1].f = 1; roundRobin[1,1].s = 2;
   forall(p in Periods :  p > 1) {
      roundRobin[1,p].f = p+1; roundRobin[1,p].s = nbTeams - (p-2);
   };
   forall(w in Weeks:  w > 1) {
      forall(p in Periods) {
         if roundRobin[w-1,p].f <> 1 then
            if roundRobin[w-1,p].f = nbTeams then roundRobin[w,p].f = 2
            else roundRobin[w,p].f = roundRobin[w-1,p].f + 1 endif
         else
            roundRobin[w,p].f = roundRobin[w-1,p].f;
         endif;
         if roundRobin[w-1,p].s = nbTeams then roundRobin[w,p].s = 2
         else roundRobin[w,p].s = roundRobin[w-1,p].s + 1 endif;
      }
   };
   forall(w in Weeks, p in Periods)
      if roundRobin[w,p].f < roundRobin[w,p].s then
         roundRobin[w,p].g = nbTeams*(roundRobin[w,p].f-1) + roundRobin[w,p].s
      else
         roundRobin[w,p].g = nbTeams*(roundRobin[w,p].s-1) + roundRobin[w,p].f
      endif;
};
```

Figure 13 A Round-Robin Model for the Sport-Scheduling Problem (Part I)

Assuming that n denotes the number of teams, the basic idea is to fix the set of games of the first week as

$$\langle 1,2\rangle \cup \{\langle p+1, n-p+2\rangle \mid p>1\}$$

where p is a period identifier. Games of the subsequent weeks are computed by transforming a tuple $\langle f, s\rangle$ into a tuple $\langle f', s'\rangle$ where

$$f' = \begin{cases} 1 & \text{if } f=1 \\ 2 & \text{if } f=n \\ f+1 & \text{otherwise} \end{cases}$$

and

$$s' = \begin{cases} 2 & \text{if } s=n \\ s+1 & \text{otherwise} \end{cases}$$

This round-robin schedule is computed in the `initialize` instruction and the last instruction

```
{int} domain[w in Weeks] = { roundRobin[w,p].g | p in Periods };

var Teams team[Periods,EWeeks,Slots];
var Games game[Periods,Weeks];

solve {
   forall(p in Periods & w in Weeks)
      game[p,w] in domain[w];
   forall(w in EWeeks)
      alldifferent( all(p in Periods & s in Slots) team[p,w,s]);
   alldifferent(game);
   forall(p in Periods)
      cardinality(occur,values,all(w in EWeeks & s in Slots) team[p,w,s]);
   forall(p in Periods & w in Weeks)
      link(team[p,w,home],team[p,w,away],game[p,w]);
};

search {
   forall(p in Periods) {
      generateSeq(game[p]);
      forall(po in Periods :  po > 1)
         generate(game[po,p]);
   };
};
```

Figure 14 A Round-Robin Model for the Sport-Scheduling Problem (Part II)

computes the game associated with the teams. The instruction

```
{int} domain [w in Weeks]
      = { roundRobin[w,p].g | p in Periods };
```

defines the games played in a given week. This array is used in the constraint

```
game[p, w] in domain[w];
```

which forces the game variables of period `p` and of week `w` to take a game allocated to that week.

The model also contains a novel search procedure that consists of generating values for the games in the first period and in the first week, then in the second period and the second week, and so on. Table 15 depicts the experimental results for various numbers of teams. It is possible to improve the model further by exploiting even more symmetries: See (Régin 1998) for complete details.

nb. of teams	14	16	18	20	22	24	26	28	30
CPU Time (sec.)	3.91	4.97	1.00	6.41	10.36	11.81	45.66	36.2	42.38

Figure 15 Experimental Results for the Sport-Scheduling Problem

5. An Overview of Bound and Cut Cooperation

Section 2 introduced the concept of redundant modeling, i.e., the idea of using several different models of the same problem simultaneously. Redundant modeling offers a natural integration scheme for constraint and integer programming: simply use the constraint and the integer models together to approach the application at hand. More precisely, redundant modeling for cooperation is based on the idea is of relaxing all linear constraints (i.e., to remove their integrality constraints) of the integer-programming model and considering the resulting linear program as a global constraint. This global constraint is referred to as the *LP-constraint* below. This section gives an overview of two cooperation schemes based on redundant modeling: bound and cut cooperation. Observe that redundant modeling is not the only way to integrate the two technologies. Decomposition methods such as column generation (e.g., Junker et al. 1999) and Benders decomposition (e.g., Jain and Grossman 2001) are very interesting alternatives that are not discussed in this paper.

5.1. Bound Cooperation

In bound cooperation (e.g., Beringer and De Backer 1994), the LP-constraint follows the traditional constraint programming protocol. It is handled independently using a specialized algorithm and only interacts with the other constraints through the domain store, i.e., by updating the domain of the variables. (Note that, in optimization problems, constraint programming has an objective-function variable to represent the values of the objective function.)

More precisely, the LP-constraint is invoked by the constraint-programming system once initially and, then, each time the bound of one of its variables is updated. Each time the LP-constraint is invoked (except for the first time), the linear program is reoptimized to integrate the new bound information (e.g., using the dual simplex). The role of the LP-constraint is thus to detect infeasibility or suboptimality and to update the domain store with a new lower bound on the objective-function variable as well as any other bound information that was obtained, e.g., through reduced cost fixing. These new bounds can be then used by other constraints to make additional deductions that may be transmitted again to the LP-constraint until no further deductions are made. Note that this process occurs, not only at the root node, but at every node of the search tree.

The benefits of bound cooperation come from the fact that the domain and bound reduction produced by the constraint-programming solver are now available to the linear-programming solver, thus improving its lower bound. Similarly, the constraint-programming solver benefits from the lower bound and may filter the domain store further. Bound cooperation is illustrated on a scene-allocation problem in Section 6.

5.2. Cut Cooperation

Cut cooperation is a significant extension of bound cooperation, which is now receiving increased attention (e.g., Hooker et al. 2001, Refalo 1999). Once again, its basic idea is a global LP-constraint. However, contrary to bound cooperation, cut cooperations updates the LP-constraint during the execution to integrate information from the constraint-programming and the LP solvers. More precisely, cut cooperation is based on extending the role of global constraints in constraint-programming systems. A global constraint C in cut cooperation is now associated with two algorithms:

1. a filtering algorithm that, given a domain store, produces a new domain removing values that are inconsistent with C;
2. a cut-generation algorithm that, given a domain store D, generates a linearization of $C \wedge D$.

The filtering algorithm is the standard constraint-programming requirement. The cut-generation algorithm is responsible for producing, given the current domain store and, possibly, the current solution to the LP-constraint, a linear formulation of $C \wedge D$.

Given this view of global constraints, cut cooperation can then be characterized as follows. It starts from the constraint model model and generates a linear formulation of each constraint, using the domain store and a common set of integer-programming variables. The linear formulation of all constraints are clustered together to produce the LP-constraint and traditional bound cooperation takes place. In addition, each time new bounds are available from the constraint-programming solver for a global constraint C and/or each time a new linear programming solution is generated, the cut-generation algorithm is invoked to produce a new LP-constraint. In practice, of course, the cut-generation algorithm does not generate a new linearization at each step but, rather, it updates the previously generated formulations incrementally by generating local cuts or by fixing variables.

Ideally, the linear formulation of $C \wedge D$ should be as sharp as possible, i.e., it should be the convex hull of the solutions of $C \wedge D$. For some global constraints (e.g., piecewise linear constraints), it is not difficult to compute and update the convex hull. For some others (e.g., the `circuit` constraint that holds if a set of variables represents a Hamiltonian circuit), there exists no compact representation of the convex hull. In these cases, it is possible to add local cuts based on the domain store and the current LP solution as is typical in branch-and-cut algorithms (Hoffman and Padberg 1985). Finally, for some global constraints, although a sharp formulation may exist, it may not be desirable because it may represent a poor tradeoff between the time spent pruning and searching.

One of the advantages of cut cooperation is to make the problem substructure explicit so that polyhedral cuts can be generated appropriately. The structure can also be used to design more effective branching schemes, an idea that is at the core of constraint-programming systems. It is also present in integer-programming systems (e.g., SOS constraints; Beale and Forrest 1976), but to a lesser extent.

Cut cooperation is now an active research topic and may yield significant benefits for some classes of applications. It has only been explored in very limited ways at this point. Section 7 describes its application to a transportation problem.

6. Scene Allocation

This section considers a scene-allocation model that combines interesting feasibility constraints with an objective function. The problem and the models we describe raise several modeling issues and illustrate the benefits of a constraint and integer programming cooperation. Section 6.1 specifies the problem. Section 6.2 presents a simple constraint model. Section 6.3 shows how to remove symmetries. Section 6.4 illustrates how constraint and integer programming may cooperate using a redundant modeling approach.

6.1. Problem Description

The scene-allocation problem was communicated to us by Irvin Lustig, who also provided an integer-programming model and a specific instance. It consists of deciding when to shoot scenes for a movie. Each scene invoves a number of actors and at most 5 scenes a day can be filmed. All actors of a scene must, of course, be present on the day the scene is shot. The actors have fees representing the amount to be paid per day they spent in the studio. The goal of the application is to minimize the production costs.

6.2. A Simple Constraint-Programming Model

Figure 16 describes a simple constraint program to solve the scene shooting problem, while Figure 17

```
int maxScene = ...;
range Scene 1..maxScene;
int maxDay = ...;
range Day 1..maxDay;
enum Actor = ...;
int pay[Actor] = ...;
{Actor} appears[Scene] = ...;

{Scene} which[a in Actor] = { i | i in Scene :  a in appears[i] };
var Day shoot[Scene];

minimize
   sum(a in Actor, d in Day) pay[a] * (or(s in which[a]) shoot[s] = d)
subject to
   atmost(all(i in Day) 5,all(i in Day) i,shoot);

search {
   forall(s in Scene
         ordered by increasing <dsize(scene[s]),-sum(a in appears[s]) pay[a]>)
      tryall(d in Day:  isInDomain(shoot[s],d))
         shoot[s] = d;
};
```

Figure 16 **The Scene Allocation Model**

```
maxScene = 19;
maxDay = 5;
Actor = {
   Patt, Casta, Scolaro, Murphy, Brown, Hacket, Anderson,
   McDougal, Mercer, Spring, Thompson
};
pay = #[
   Patt :  26481, Casta:  25043, Scolaro:  30310,
   Murphy:  4085, Brown:  7562, Hacket:  9381,
   Anderson:  8770, McDougal:  5788, Mercer:  7423,
   Spring:  3303, Thompson:  9593
]#;
appears = [
   { Hacket },
   { Patt, Hacket, Brown, Murphy },
   { McDougal, Scolaro, Mercer, Brown },
   { Casta, Mercer },
   { Mercer, Anderson, Patt, McDougal, Spring },
   { Thompson, McDougal, Anderson, Scolaro, Spring },
   { Casta, Patt },
   { Mercer, Murphy },
   { Casta, McDougal, Mercer, Scolaro, Thompson },
   { Casta, McDougal, Scolaro, Patt },
   { Patt },
   { Hacket, Thompson, McDougal, Murphy, Brown },
   { Hacket, Murphy, Casta, Patt },
   { Anderson, Scolaro },
   { Thompson, Murphy, McDougal, Patt },
   { Scolaro, McDougal, Casta, Mercer },
   { Scolaro, Patt, Brown },
   { Scolaro, McDougal, Hacket, Thompson },
   { Casta }
];
```

Figure 17 Instance Data for the Scene-Allocation Problem

depicts the instance data. The model starts by defining the ranges for the scenes and the days, the set of actors and their daily fees, and the set of actors who appear in each of the scenes. The declaration

```
{Scene} which[a in Actor]
  = { i | i in Scene: a in appears[i] };
```

computes the scenes in which an actor appears. The decision variables associate with every scene `s` its shooting day `shoot[s]`. The objective function computes the sum of all fees. The expression

```
        or(s in which[a]) shoot[s] = d
```

denotes whether actor `a` is present on day `d`. The only constraint of the model

```
atmost (all(i in Day) 5, all(i in Day) i,shoot);
```

is a global cardinality constraint expressing that at most 5 scenes a day can be shot. The search procedure selects the scenes in sequence, using the *first-fail principle* and breaking ties by selecting first the most expensive scenes. It then assigns a shooting day to the selected scene.

This model solves the instance depicted in 17 in about 421 seconds and with about 650,000 nodes. The integer-programming model in OPL (The OPL implementation uses a state-of-the-art MIP system.) solves the problem in about 100 seconds. There are two reasons for the poor performance of the constraint-programming model:

1. the problem has many symmetries, increasing the size of the search space;
2. the lower bound computed by constraint programming is extremely poor in this model.

We show how to remedy both limitations in the rest of this section.

6.3. Removing Symmetries

As mentioned, the scene-allocation problem has many symmetries. In particular, the exact shooting day of a scene is not significant. What really matters is how the scenes are clustered, not the particular days to which they are assigned. However, it is not easy to state constraints removing these symmetries, mainly because of the interaction with the search procedure.

As a first approximation, let us ignore the search procedure. A step towards the solution then consists of observing what happens when the scenes are assigned a day in sequence. It is perfectly valid to assign the first scene on day 1, since the days are all equivalent at that stage. For the second scene, only two days must be considered: either the scene is shot on the same day as scene 1 (day 1) or it is scheduled on another day, say day 2. There is no need to consider days 3–5, since they are equivalent to day 2 for the purpose of this application. The same reasoning can be generalized to any scene. If days $1 \ldots d$ have been assigned for the first $n-1$ scenes, then scene n needs only to consider days $1 \ldots d+1$, i.e., the days where some scenes are already allocated and one additional new day, if any, are left. It is thus possible to write the constraints

```
shoot[1] = 1;
forall(s in Scene: s > 1)
   shoot[s] <= 1 + max (p in 1..s - 1) shoot[p];
```

The problem with these constraints is that they impose an implicit ordering of scenes that interferes with the dynamic ordering of the search procedure. Indeed, if $< s_1, \ldots, s_n >$ is the dynamic ordering chosen by the search procedure, then the set of constraints should really enforce

```
shoot[s1] = 1
shoot[s2] <= 1 + shoot[s1]
shoot[s3] <= 1 + max(shoot[s1], shoot[s2])
...
```

Since this ordering is dynamic, it is not easy to enforce these constraints statically. (As observed by one of the reviewers, it is possible to do so by representing the sequence in which scenes are assigned explicitly. However, this approach also requires assigning the sequence variables during search in order to benefit fully from the symmetry removal. As a consequence, this approach is essentially similar to what is presented here.) Fortunately, it is simple to generate these constraints dynamically during search. Figure 18 depicts a model that removes symmetries. The main novelty is the declaration of the search procedure

```
SearchProcedure label(int day)
{
  select(s in Scene: not bound(shoot[s])
      ordered by increasing <dsize(shoot[s]),
        -sum(a in appears[s]) pay[a]>)
    tryall(d in Day: d <= day + 1 & isInDomain
        (shoot[s], d)) {
      shoot[s] = d;
      if d = day + 1 then
        label(d)
      else
        label(day)
      endif;
  };
};
```

whose formal parameter denotes the number of days allocated so far to the scenes. The procedure first selects, in a deterministic way, a scene that is not yet assigned (i.e., `not bound(shoot[s])`) using the same heuristic as before. Once this scene is selected, the procedure tries to assign the scene a day from the set `1..day+1`, i.e., the days already allocated and one new day (if available). These are the only days that need to be considered since the other "new" days are "equivalent" to `day + 1`. The `tryall` instruction contains a filter `d <= day + 1` for that purpose. The search procedure then is called recursively, possibly with a new

```
int maxScene = ...;
range Scene 1..maxScene;
int maxDay = ...;
range Day 1..maxDay;
enum Actor = ...;
int pay[Actor] = ...;
{Actor} appears[Scene] = ...;

{Scene} which[a in Actor] = { i | i in Scene :  a in appears[i] };
var Day shoot[Scene];

SearchProcedure label(int day)
{
   select(s in Scene:  not bound(shoot[s])
         ordered by increasing <dsize(shoot[s]),-sum(a in appears[s]) pay[a]>)
      tryall(d in Day:  d <= day + 1 & isInDomain(shoot[s],d)) {
         shoot[s] = d;
         if d = day + 1 then
            label(d)
         else
            label(day)
         endif;
      };
};

minimize
   sum(a in Actor, d in Day) pay[a] * (or(s in which[a]) shoot[s] = d)
subject to
   atmost(all(i in Day) 5,all(i in Day) i,shoot);

search label(0);
```

Figure 18 The Scene-Allocation Model with Symmetry Removal

value for the parameter if the new day was allocated to the scene. The model search procedure simply calls `label(0)` to generate a day to all the scenes, since no days are allocated initially.

It is interesting to note that this model now solves the instance in 6.5 seconds using around 11,000 nodes. However, as mentioned, it still computes a poor lower bound. The next section addresses this issue.

6.4. Redundant Modeling

To improve the lower bound of the previous models, Figure 19 uses a redundant modeling approach that includes both the integer and constraint programming formulations. Consider the novelties of this model. The first addition

```
var int isPaid[Actor,Day] in 0..1;
var int isShot[Scene,Day] in 0..1;
```

is the declaration of the integer-programming variables that specify whether an actor is paid on a given day and whether a scene is filmed on a given day. The second novelty is the objective function

```
sum(a in Actor, d in Day) pay[a] * isPaid[a,d]
```

which comes from the integer-programming formulation. The last novelty is the constraint statement that now includes the existing constraints as well as the integer-programming constraints, and the link constraints

```
forall(s in Scene)
    isShot[s,shoot[s]] = 1;
```

```
int maxScene = ...; range Scene 1..maxScene;
int maxDay = ...; range Day 1..maxDay;
enum Actor = ...;
int pay[Actor] = ...;
{Actor} appears[Scene] = ...;
{Scene} which[a in Actor] = { i | i in Scene :  a in appears[i] };

var Day shoot[Scene];
var int isPaid[Actor,Day] in 0..1;
var int isShot[Scene,Day] in 0..1;

SearchProcedure label(int day) {
   select(s in Scene:  not bound(shoot[s])
         ordered by increasing <dsize(shoot[s]),-sum(a in appears[s]) pay[a]>)
      tryall(d in Day:  d <= day + 1 & isInDomain(shoot[s],d)) {
         shoot[s] = d;
         if d = day + 1 then label(d) else label(day) endif;
      };
};

minimize with linear relaxation
   sum(a in Actor, d in Day) pay[a] * isPaid[a,d]
subject to {
   atmost(all(i in Day) 5,all(i in Day) i,shoot);

   forall(s in Scene, a in appears[s], d in Day)
      isShot[s,d] <= isPaid[a,d];
   forall(s in Scene)
      sum(d in Day) isShot[s,d] = 1;
   forall(d in Day)
      sum(s in Scene) isShot[s,d] <= 5;

   forall(s in Scene)
      isShot[s,shoot[s]] = 1;
};
search label(0);
```

Figure 19 The Scene-Allocation Model with Redundant Modeling and Cooperation

A constraint

```
isShot[s,shoot[s]] = 1;
```

in fact connects the constraint variable `shoot[s]` to the integer-programming variables `isShot[s,1]`, ..., `isShot[s,5]`. Observe that these constraints feature the `element` constraint, i.e., the ability to index an array of variables with a variable. The `element` constraint performs a variety of deductions. In particular, it makes sure that, when `shoot[s]` is given a value in the search procedure, the corresponding integer variables in array `isShot` are given the appropriate values. The ability to index arrays by expressions containing variables is called an `element` constraint for historic reasons. This functionality originated in the CHIP system (Van Hentenryck 1987b, 1989, Van Hentenryck and Carillon 1988) which is a constraint logic programming language without arrays but with lists. CHIP featured a constraint `element(i,L,e)` that holds if `e` is element `i` of list `L`. The `element` constraint is generally considered as the first global constraint since it captures a disjunction of more primitive constraints. Indeed, if `L` is the list `[v`$_1$`,...,v`$_n$`]`, the `element` constraint can be specified as

$$(i=1 \ \& \ e=v_1) \vee \cdots \vee (i=n \ \& \ e=v_n).$$

The solver may enforce arc consistency or other forms of consistency on the `element` constraint. Expressions indexing arrays by variables are in fact transformed into `element` constraints when the expressions are parsed or constructed.

Note also the keyword `with linear relaxation` that specifies that OPL should use bound cooperation to solve this model. This keyword specifies that all linear constraints should be sent both to the constraint- and linear-programming solvers.

It is interesting to mention the performance of the redundant model for the scene-allocation problem. The instance is now solved in 5.9 seconds using only around 700 nodes. Observe the significant reduction in the number of nodes and the progress achieved compared to the original constraint- and integer-programming models.

7. Transportation

This section describes a cut-cooperation approach to a transportation application, where the cost is given by a summation of non-convex piecewise linear functions of the individual variables. Section 7.1 describes the problem and the OPL model. Section 7.2 describes the cut-cooperation approach on this model.

7.1. The Problem and the OPL Model

Given a set of suppliers and a set of customers, the problem consists of deciding how much to ship from the suppliers to the customers so that the problem constraints are satisfied and the shipment cost is minimized. The constraints are of two types: (1) the supplier constraints specify that the entire supply must be shipped, and (2) the customer constraints specify that the demand must be met. The objective function is a summation of the cost of the individual shipments from a given supplier to a given customer. However, the individual cost of shipment is a piecewise-linear function of the shipment. Figure 20 describes an OPL model for a specific instance of this problem. It starts by defining the set of customers and the set of suppliers. It then specifies the supplies and the demand and declares the decision variables: variable `x[s,c]` represents the shipment from supplier `s` to customer `c`. The problem constraints are typical from transportation problems. More interesting is the objective function, which is a summation of the piecewise linear function of the shipments. The first two lines are OPL commands to specify that a cut-cooperation scheme should be applied on the piecewise constraints.

7.2. Cut Cooperation

An Integer-Programming Formulation. We first describe a traditional integer-programming formulation of this model (e.g., Fourer 1992). Figure 21 depicts a reformulation of the statement to isolate the piecewise linear functions in constraints of the form

```
y[i,j] = piecewise { 120 -> 200 ; 80
                     -> 400 ; 50 } x[i,j];
```

To obtain an integer program, it remains to show how to linearize constraints of the form

$$y = f(x)$$

where f is a piecewise-linear function. Let us assume that f is described by its breakpoints $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$ as well as by its initial and final slopes σ_0 and σ_{n+1}. If $x \in [x_1, x_n]$, then the linearization of $y = f(x)$ consists of taking the convex hull of the breakpoints, which produces the linear constraints

$$\begin{aligned}
&x = \lambda_1 x_1 + \cdots + \lambda_n x_n\\
&y = \lambda_1 f(x_1) + \cdots + \lambda_n f(x_n)\\
&\lambda_1 + \cdots + \lambda_n = 1\\
&\lambda_i \geq 0 \ \ (1 \leq i \leq n)
\end{aligned}$$

where $\lambda_1, \ldots, \lambda_n$ are brand new variables. In the general case, the linear constraints become

$$\begin{aligned}
&x = -\lambda_0 + \lambda_1 x_1 + \cdots + \lambda_n x_n\\
&y = -\lambda_0\sigma_0 + \lambda_1 f(x_1) + \cdots + \lambda_n f(x_n) + \lambda_{n+1}\sigma_{n+1}\\
&\lambda_1 + \cdots + \lambda_n = 1\\
&\lambda_i \geq 0 \ \ (0 \leq i \leq n+1).
\end{aligned}$$

In order to obtain an integer program that correctly represents the piecewise linear function, it is also necessary to specify that at most two successive λ_i variables can be non-zero. Integer-programming solvers

```
setting MIPmethod = SOLVERmip;
setting PiecewiseCuts = 1;

int nbCustomers = 6;
int nbSuppliers = 5;
range Customers 1..nbCustomers;
range Suppliers 1..nbSuppliers;

float supply[Suppliers] = [ 1000, 850, 1250, 750, 2250];
float demand[Customers] = [ 900, 1200, 600, 400, 2500, 500 ];

var float+ x[Suppliers,Customers];

minimize
   sum(i in Suppliers, j in Customers)
      piecewise { 120 -> 200 ; 80 -> 400 ; 50 } x[i,j]
subject to {
   forall(i in Suppliers)
      sum(j in Customers) x[i,j] = supply[i];

   forall(j in Customers)
      sum(i in Suppliers) x[i,j] = demand[j]; };
```

Figure 20 The Transportation Model

```
setting MIPmethod = SOLVERmip;
setting PiecewiseCuts = 1;

int nbCustomers = 6;
int nbSuppliers = 5;
range Customers 1..nbCustomers;
range Suppliers 1..nbSuppliers;

float supply[Suppliers] = [ 1000, 850, 1250, 750, 2250];
float demand[Customers] = [ 900, 1200, 600, 400, 2500, 500 ];

var float+ x[Suppliers,Customers];
var float+ y[Suppliers,Customers];

minimize
   sum(i in Suppliers, j in Customers) y[i,j]
subject to {
   forall(i in Suppliers, j in Customers)
      y[i,j] = piecewise { 120 -> 200 ; 80 -> 400 ; 50 } x[i,j];

   forall(i in Suppliers)
      sum(j in Customers) x[i,j] = supply[i];

   forall(j in Customers)
      sum(i in Suppliers) x[i,j] = demand[j]; };
```

Figure 21 The Transportation Model Reformulated

typically impose this requirement by special ordered sets (SOS) constraints that are used for branching during search (Beale and Forrest 1976). The linear relaxation simply ignores these constraints.

Cooperation. We now turn to the cut-cooperation scheme that, when applied to this problem, illustrates well the synergy between integer and constraint programming (Refalo, 1999). The OPL implementation uses global constraints of the form `y = f(x)`, where `f` is a piecewise-linear function. These constraints support a filtering algorithm for reducing the domains of variables appearing in piecewise constraints. Initially, the model is sent to the constraint-programming solver and its linearization becomes a global LP-constraint. The linearization is obtained by linearizing the piecewise constraints as discussed earlier. Traditional bound cooperation then takes place, during which the constraint-programming solver reduces the bounds of the variables and the LP-constraint computes a lower bound. Moreover, at each node of the search tree, the bounds produced by the constraint-programming solver are used to produce new convex hulls for each of the piecewise linear constraints and the relevant part of the domain store. Consider for instance the piecewise-linear function $y = f(x)$ shown in Figure 22 and assume that the range of x is $[0, M]$. Figure 23 depicts the convex hull of the constraints $x \in [0, M]$ and $y = f(x)$. Now, assume that the constraint-rogramming solver reduces the bounds of x to $[l, u]$. A bound cooperation scheme would take the intersection of the original convex hull to produce the region depicted in Figure 24 and the LP-constraint would produce a lower bound based on that feasible region.

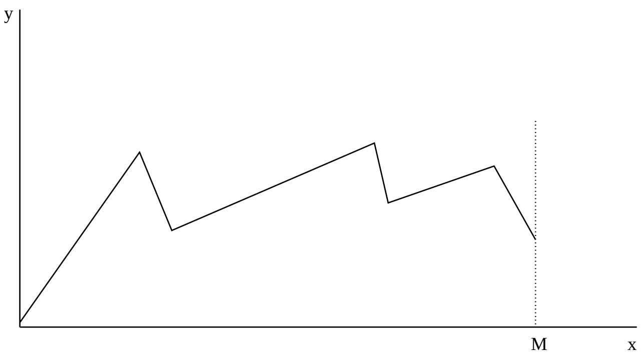

Figure 22 A Piecewise-Linear Function

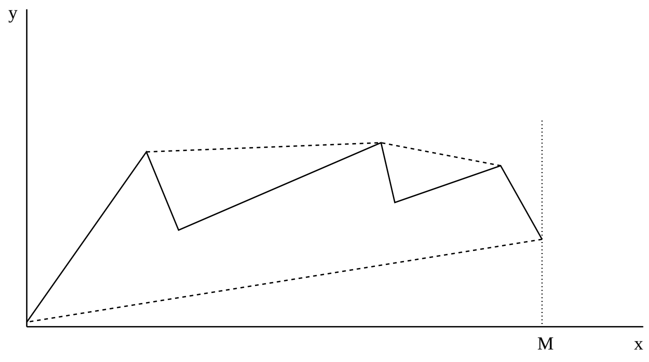

Figure 23 The Initial Convex Hull of the Piecewise-Linear Function with the Domains

A cut-ooperation scheme, however, generates a new convex hull by exploiting the new bound information to produce the region depicted in Figure 25. Observe that this region may be substantially smaller than the one produced by a bound-cooperation scheme and hence the LP-constraint may produce tighter lower bounds. Note that the lower bound computed by the LP-constraint is also sent to the constraint-programming solver, which may deduce new bounds for some variables. Of course, the convex hull is not recomputed from scratch; rather the original convex hull is dynamically updated by fixing the λ_i variables and generating new cuts.

The benefits of cut cooperation are not limited to improving the lower bound. By preserving the structure of the piecewise-linear constraints, it is possible

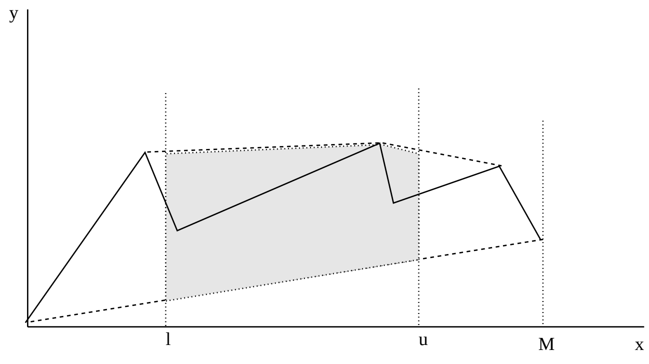

Figure 24 The Convex Hull of the Piecewise-Linear Function with the Domains After Bound Cooperation

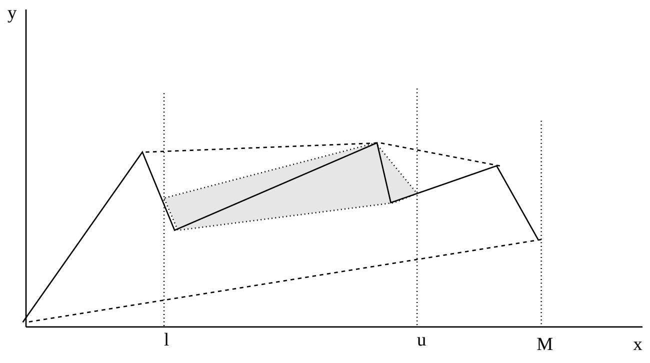

Figure 25 **The Convex Hull of the Piecewise-Linear Function with the Domains After Cut Cooperation**

to design branching schemes exploiting the combinatorial structure of the problem directly. The idea is to select a piecewise-linear function such that $y^* \neq f(x^*)$, where x^* and y^* are the current values of x and y in the LP relaxation. In fact, it is generally a good heuristic to select the constraint maximizing $|y^* - f(x^*)|$, which is another application of the first-fail principle. Branching can then proceed by creating the nodes $x < S$ and $x \geq S$, where S is the x-coordinate of the starting point of the segment that corresponds to the value of x^*.

Cut cooperation has produced order-of-magnitude improvements in efficiency for transportation problems that use piecewise-linear functions to model economies of scale (Refalo, 1999). For eaxmple, on the instance data, OPL would run out of memory when not using cooperation and it would not return after 20 minutes with a bound-cooperation scheme. A cut-cooperation scheme would yield the optimal solution in about 2 seconds. See Refalo (1999) for extensive experimental results on this and similar problems.

8. Conclusion

In recent years, it has been increasingly recognized that integer programming and constraint programming have orthogonal and complementary strengths in stating and solving combinatorial optimization problems and much effort is being devoted to finding natural ways of exploiting their respective advantages.

The optimization programming language OPL originated from an attempt to integrate constraint and integer programming, both at the language and at the solver levels. From a language standpoint, OPL shares high-level set and algebraic expressions with mathematical modeling languages, and a rich constraint language and the ability to specify search procedures with constraint-programming languages. From an implementation standpoint, OPL makes it possible to use both constraint and mathematical-programming tools to solve an application. Cooperation is achieved through redundant modeling, i.e., by using both an integer-programming and a constraint-programming formulation. OPL supports bound cooperation, where the two solvers cooperate by exchanging bound information. It also supports, in a very limited way, cut cooperation. These collaboration schemes exploit the strengths of constraint programming to rule out infeasible solutions and the strengths of integer programming for computing lower bounds in minimization problems.

The integration of constraint and integer programming is a promising research area. By making substructures explicit in modeling, constraint programming offers a nice framework for integrating both technologies. Moreover, the different foci of both technologies, feasibility versus optimality, and the different underlying algorithms, filtering algorithms versus lower bounding techniques, make it likely that a tight integration as proposed by cut cooperation will produce interesting improvements on some classes of applications. Our future research will aim at studying these issues in more detail.

Acknowledgments
Special thanks to the reviewers whose comments and suggestions improved the presentation substantially.

References

Beale, E., J. Forrest. 1976. Global optimization using special ordered sets. *Mathematical Programming* **10** 52–69.

Beldiceanu, N., E. Contejean. 1994. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling* **20** 97–123.

Beringer, H., B. De Backer. 1994. Combinatorial Problem Solving in Constraint Logic Programming with Cooperating Solvers. *Logic Programming: Formal Methods and Practical Applications*, Elsevier Science Publishers.

Benhamou, F., W. Older. 1997. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming* **32** 1–24.

Berthier, F. 1988. Using CHIP to support decision making. *Actes du Séminaire 1988—Programmation en Logique*, Trégastel, France.

Bisschop, J., A. Meeraus. 1982. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study* **20** 1–29.

Bockmayer, A., T. Kasper. 1998. Branch and infer: a unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing* **10** 287–300.

Borning, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transaction on Programming Languages and Systems* **3** 353–387.

Caseau, Y., F. Laburthe. 1997. Solving various weighted matching problems with constraints. *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*. Lintz, Austria. 17–31.

Cheng, B. M. W., J. H. M. Lee, H. F. Leung, Y. W. Leung. 1996. Speeding up constraint propagation by redundant modeling. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, Cambridge, MA. Springer Verlag. 104–118.

Colmerauer, A. 1982. PROLOG II: manuel de référence et modèle théorique. Technical report, GIA—Faculté de Sciences de Luminy, Marseilles, France.

Colmerauer, A. 1990. An introduction to Prolog III. *Commun. ACM* **28** 412–418.

Colmerauer, A. 1996. Spécification de Prolog IV. Technical report, Laboratoire d'informatique de Marseilles, Marseilles, France.

Colmerauer, A., H. Kanoui, R. Pasero, P. Roussel. 1973. Un système de communication homme-machine en Français. Rapport de recherche, Groupe Intelligence Artificielle, Université d'Aix-Marseilles II, Marseilles, France.

Costa, M. C. 1994. Persistency in maximum cardinality bipartite matching. *Operations Research Letters* **15** 143–149.

de Farias, I. R., E. L. Johnson, G. L. Nemhauser. 2001. Branch-and-cut for combinatorial optimization problems without auxiliary binary variables. *Knowledge Engineering Review* to appear.

Dincbas, M., H. Simonis, P. Van Hentenryck. 1988. Solving the car sequencing problem in constraint logic programming. *European Conference on Artificial Intelligence (ECAI-88)*. Munich, Germany.

Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier. 1988. The constraint logic programming language CHIP. *Proceedings of the International Conference on Fifth Generation Computer Systems*. Tokyo, Japan.

El Sakkout, H., M. Wallace. 2000. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints* **5** 359–388.

Fourer, R. 1992. Solving piecewise-linear programs: experiments with a simplex approach. *ORSA Journal on Computing* **4** 16–31.

Fourer, R., D. Gay, B. W. Kernighan. 1993. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA.

Gervet, C. 1994. Conjunto: constraint logic programming with finite set domains. *Proc. of the International Symposium on Logic Programming (ILPS'94)*, Ithaca, NY.

Gotlieb, A., B. Botella, M. Rueher. 2000. A CLP framework for computing structural test data. *Proceedings of the First International Conference on Computational Logic*, London, UK.

Graf, T., P. Van Hentenryck, C. Pradelles, L. Zimmer. 1989. Simulation of hybrid circuits in constraint logic programming. *International Joint Conference on Artificial Intelligence*, Detroit, MI.

Harvey, W. D., M. L. Ginsberg. 1995. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada.

Heintze, N. C., S. Michaylov, P. J. Stuckey. 1987. CLP($\Re$) and some electrical engineering problems. *Proceedings of the 4th International Conference on Logic Programming*, MIT Press, Melbourne, Australia.

Hoffman, K., M. W. Padberg. 1985. LP-based combinatorial problem solving. *Annals of Operations Research* **4** 145–194.

Hooker, J. N. 2000. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. John Wiley and Sons, New York.

Hooker, J. N., G. Ottosson, E. S. Thornsteinsson, H.-J. Kim. 2001. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review* to appear.

Huynh, T., C. Lassez. 1988. A CLP($\Re$) option trading analysis system. *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Seattle, WA.

Jaffar, J., J.-L. Lassez. 1987. Constraint logic programming. *POPL-87*, Munich, Germany.

Jaffar, J., M. Maher. 1994. Constraint logic programming: a survey. *Journal of Logic Programming* **19/20** 503–582.

Jain, V., I. E. Grossmann. 2001. Algorithms for hybrid MILP/CLP models for a class of optimization problems. *INFORMS Journal on Computing* **13** 258–276.

Junker, U., S. Karish, N. Kohl, B. Vaaben, T. Fahle, M. Sellman. 1999. A framework for constraint programming based column generation. *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*, Alexandria, VA.

Kanellakis, P. C., G. M. Kuper, P. Z. Revesz. 1990. Constraint query languages. *PODS-90*, Nashville, TN.

Kohn, W., A. Nerode, V. S. Subrhamanian. 1995. Constraint logic programming: hybrid control and logic as linear programming. V. Saraswat and P. Van Hentenryck, eds. *Principles and Practice of Constraint Programming*, MIT Press, Cambridge, MA, 85–100.

Kowalski, R. 1974. Predicate logic as programming language. *Proceedings of the IFIP Congress 74*, 569–574.

Laburthe, F., Y. Caseau. 1998. SALSA: a language for search algorithms. *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy.

Mackworth. A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* **8** 99–118.

Maher, M. J. 1987. Logic semantics for a class of committed-choice programs. *Fourth International Conference on Logic Programming*, Melbourne, Australia, 858–876.

Marriott, K., P. Stuckey. 1999. *Programming with Constraints*. The MIT Press, Cambridge, MA.

McAloon, K. C. Tretkoff. 1995. `2LP`: linear programming and logic programming. V. Saraswat and P. Van Hentenryck, eds, *Principles and Practice of Constraint Programming*, MIT Press, Cambridge, MA.

McAloon, K., C. Tretkoff, G. Wetzel. 1997. Sport league scheduling. In *Proceedings of the 3th Ilog International Users Meeting*, Paris, France.

Montanari, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. *Information Science* **7** 95–132.

Nuijten, W. 1994. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. PhD thesis, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

Perron, L. 1999. Search procedures and parallelism in constraint programming. *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*, Alexandria, VA.

Pesant, G., M. Gendreau, J.-M. Rousseau. 1997. GENIUS-CP: a generic single-vehicle routing algorithm. *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*, Lintz, Austria, 420–434.

Puget, J.-F., M. Leconte. 1995. Beyond the glass box: constraints as objects. *Proceedings of the International Symposium on Logic Programming (ILPS-95)*, Portland, OR.

P. Réfalo. 1999. Tight cooperation and its application in piecewise linear optimization. In *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*, Alexandria, VA.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, Washington, 362–367.

Régin, J.-C. 1996. Generalized arc consistency for global cardinality constraint. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, 209–215.

Régin, J.-C. 1998. Sport league scheduling. *INFORMS National Meeting*, Montreal, Canada.

Saraswat, V. A. 1989. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.

Savelbergh, M. W. P. 1994. Preprocessing and probing for mixed integer programming problems. *ORSA Journal of Computing* **6** 445–454.

Schulte, C. 1997. Programming constraint inference engines. *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330, Linz, Austria, October 1997. Springer-Verlag. 519–533.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, 417–431.

Simonis, H., M. Dincbas. 1987. Using an extended prolog for digital circuit design. *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, Munich, Germany, 165–188.

Smolka, G. 1995. The Oz Programming Model. Jan van Leeuwen, ed., *Computer Science Today*. LNCS, No. 1000, Springer Verlag.

van Emden, M. H. 1999. Algorithmic power from declarative use of redundant constraints. *Constraints*, 363–381.

Van Hentenryck, P. 1987a. A Framework for Consistency Techniques in Logic Programming. *IJCAI-87*, Milan, Italy.

Van Hentenryck, P. 1987b. *Consistency Techniques in Logic Programming*. PhD thesis, University of Namur, Namur, Belgium.

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

Van Hentenryck, P. 1991. Constraint Logic Programming. *Knowledge Engineering Review* **6** 151–194.

Van Hentenryck, P. 1987. *Encyclopedia of Science and Technology*, chapter *Constraint Programming*. Marcel Dekker.

Van Hentenryck, P, 1999. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA.

Van Hentenryck, P., J.-P. Carillon. 1988. Generality Versus Specificity: An Experience with AI and OR Techniques. *Proceedings of the American Association for Artificial Intelligence (AAAI-88)*, St. Paul, MN.

Van Hentenryck, P., M. Dincbas. 1986. Domains in Logic Programming. *AAAI-86*, Philadelphia, PA.

Van Hentenryck, P., L. Michel. 2000. *OPL Script: Composing and Controlling Models*. Lecture Note in Artificial Intelligence (LNAI 1865). Springer Verlag.

Van Hentenryck, P., L. Michel, Y. Deville. 1997. *Numerica: a Modeling Language for Global Optimization*. MIT Press, Cambridge, MA.

Van Hentenryck, P., L. Michel, L. Perron, J. C. Regin. 1999. Constraint Programming in OPL. *Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP'99)*, Paris, France.

Van Hentenryck, P., L. Perron, J.-F. Puget. 2000. Search and Strategies in OPL. *ACM Transactions on Computational Logic* **1** 1–36.

Van Hentenryck, P., V. Saraswat. 1996. Strategic Directions in Constraint Programming. *ACM Computing Surveys* **28** 701–726.

Van Hentenryck, P., V. Saraswat, Y. Deville. 1995. *The Design, Implementation, and Evaluation of the Constraint Language* `cc(FD)`. *Constraint Programming: Basics and Trends*. Springer Verlag.

Waltz, D. 1972. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI271, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Wolsey, L. 1998. *Integer Programming*. John Wiley and Sons, New York, NY.

Accepted by John W. Chinneck; received November 2000; revised July 2001, November 2001; accepted November 2001.