Constraint Programming in OPL

P. Van Hentenryck[1], L. Michel[2], L. Perron[2], and J.-C. Régin[2]

[1] UCL, Place Sainte-Barbe, 2, B-1348 Louvain-La-Neuve, Belgium
[2] Ilog SA, 9 rue de Verdun, F-94253 Gentilly Cedex, France

Abstract. OPL is a modeling language for mathematical programming and combinatorial optimization problems. It is the first modeling language to combine high-level algebraic and set notations from modeling languages with a rich constraint language and the ability to specify search procedures and strategies that is the essence of constraint programming. In addition, OPL models can be controlled and composed using OPLSCRIPT, a script language that simplifies the development of applications that solve sequences of models, several instances of the same model, or a combination of both as in column-generation applications. This paper illustrates some of the functionalities of OPL for constraint programming using frequency allocation, sport-scheduling, and job-shop scheduling applications. It also illustrates how OPL models can be composed using OPLSCRIPT on a simple configuration example.

1 Introduction

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

The last two decades have witnessed substantial development in tools to simplify the design and implementation of combinatorial optimization problems. Their goal is to decrease development time substantially while preserving most of the efficiency of specialized programs. Most tools can be classified in two categories: mathematical modeling languages and constraint programming languages. Mathematical modeling languages such as AMPL [4] and GAMS [1] provides very high-level algebraic and set notations to express concisely mathematical problems that can then be solved using state-of-the-art solvers. These modeling languages do not require specific programming skills and can be used by a wide audience. Constraint programming languages such as CHIP [3], PROLOG III and its successors [2], OZ [12], and ILOG SOLVER [11] have orthogonal strenghts. Their constraint languages, and their underlying solvers, go beyond traditional linear and nonlinear constraints and support logical, high-order, and global constraints. They also make it possible to program search procedures to specify how to explore the search space. However, these languages are mostly aimed at computer scientists and often have weaker abstractions for algebraic and set manipulation.

The work described in this paper originated as an attempt to unify modeling and constraint programming languages and their underlying implementation technologies. It led to the development of the optimization programming language OPL [13], its associated script language OPLScript [?], and its development environment OPL Studio.

OPL is a modeling sharing high-level algebraic and set notations with traditional modeling languages. It also contains some novel functionalities to exploit sparsity in large-scale applications, such as the ability to index arrays with arbitrary data structures. OPL shares with constraint programming languages their rich constraint languages, their support for scheduling and resource allocation problems, and the ability to specify search procedures and strategies. OPL also makes it easy to combine different solver technologies for the same application.

OPLScript is a script language for composing and controlling OPL models. Its motivation comes from the many applications that require solving several instances of the same problem (e.g., sensibility analysis), sequences of models, or a combination of both as in column-generation applications. OPLScript supports a variety of abstractions to simplify these applications, such as OPL models as first-class objects, extensible data structures, and linear programming bases to name only a few.

OPL Studio is the development environment of OPL and OPLScript. Beyond support for the traditional "edit, execute, and debug" cycle, it provides automatic visualizations of the results (e.g., Gantt charts for scheduling applications), visual tools for debugging and monitoring OPL models (e.g., visualizations of the search space), and C++ code generation to integrate an OPL model in a larger application. The code generation produces a class for each model objects and makes it possible to add/remove constraints dynamically and to overwrite the search procedure.

The purpose of this paper is to illustrate some of the constraint programming features of OPL through a number of models. Section 2 describes a model for a frequency allocation application that illustrates how to use high-level algebraic and set manipulation, how to exploit sparsity, and how to implement search procedures in OPL. Section 3 describes a model for a sport-scheduling applications that illustrates the use of global constraints in OPL. Section 4 describes an application that illustrates the support for scheduling applications and for search strategies in OPL. Section 5 shows how OPL models can be combined using OPLScript on a configuration application. All these applications can be run on ILOG OPL Studio 2.1.

2 Frequency Allocation

The frequency-allocation problem [11] illustrates a number of interesting features of OPL: the use of complex quantifiers, and the use of a multi-criterion ordering to choose which variable to assign next. It also features an interesting data representation that is useful in large-scale linear models.

The frequency-allocation problem consists of allocating frequencies to a number of transmitters so that there is no interference between transmitters and the number of allocated frequencies is minimized. The problem described here is an actual cellular phone problem where the network is divided into cells, each cell containing a number of transmitters whose locations are specified. The interference constraints are specified as follows:

- The distance between two transmitter frequencies within a cell must not be smaller than 16.
- The distances between two transmitter frequencies from different cells vary according to their geographical situation and are described in a matrix.

The problem of course consists of assigning frequencies to transmitters to avoid interference and, if possible, to minimize the number of frequencies. The rest of this section focuses on finding a solution using a heuristic to reduce the number of allocated frequencies.

```
int nbCells = ...;
int nbFreqs = ...;
range Cells 1..nbCells;
range Freqs 1..nbFreqs;
int nbTrans[Cells] = ...;
int distance[Cells,Cells] = ...;

struct TransmitterType { Cells c; int t; };
{TransmitterType} Transmits = { <c,t> | c in Cells & t in 1..nbTrans[c] };
var Freqs freq[Transmits];

solve {
   forall(c in Cells & ordered t1, t2 in 1..nbTrans[c])
      abs(freq[<c,t1>] - freq[<c,t2>]) >= 16;

   forall(ordered c1, c2 in Cells : distance[c1,c2] > 0)
      forall(t1 in 1..nbTrans[c1] & t2 in 1..nbTrans[c2])
         abs(freq[<c1,t1>] - freq[<c2,t2>]) >= distance[c1,c2];
};

search {
   forall(t in Transmits ordered by increasing <dsize(freq[t]),nbTrans[t.c]>)
      tryall(f in Freqs ordered by decreasing nbOccur(f,freq))
         freq[t] = f;
};
```

Fig. 1. The Frequency-Allocation Problem (`alloc.mod`).

Statement 1 shows an OPL statement for the frequency-allocation problem and Statement 2 describes the instance data. Note the separation between models and data which is an interesting feature of OPL. The model data first specifies the number of cells (25 in the instance), the number of available frequencies (256 in the instance), and their associated ranges. The next declarations specify the number of transmitters needed for each cell and the distance between cells. For example, in the instance, cell 1 requires eight transmitters while cell 3 requires six transmitters. The distance between cell 1 and cell 2 is 1.

The first interesting feature of the model is how variables are declared:

```
struct TransmitterType { Cells c; int t; };
{TransmitterType} Transmits = { <c,t> | c in Cells & t in 1..nbTrans[c] };
var Freqs freq[Transmits];
```

As is clear from the problem statement, transmitters are contained within cells. The above declarations preserve this structure, which will be useful when stating constraints. A transmitter is simply described as a record containing a cell number and a transmitter number inside the cell. The set of transmitters is computed automatically from the data using

```
{TransmitterType} Transmits = { <c,t> | c in Cells & t in 1..nbTrans[c] };
```

which considers each cell and each transmitter in the cell. OPL supports a rich language to compute with sets of data structures and this instruction illustrates some of this functionality. The model then declares an array of variables

```
var Freqs freq[Transmits];
```

indexed by the set of transmitters; the values of these variables are of course the frequencies associated with the transmitters. This declaration illlustrates a fundamental aspect of OPL: arrays can be indexed by arbitrary data. In this application, the arrays of variables `freq` is indexed by the elements of `transmitters` that are records. This functionality is of primary importance to exploit sparsity in large-scale models and to simplify the statement of many combinatorial optimization problems.

There are two main groups of constraints in this model. The first set of constraints handles the distance constraints between transmitters inside a cell. The instruction

```
forall(c in Cells & ordered t1, t2 in 1..nbTrans[c])
    abs(freq[<c,t1>] - freq[<c,t2>]) >= 16;
```

enforces the constraint that the distance between two transmitters inside a cell is at least 16. The instruction is compact mainly because we can quantify several variables in `forall` statements and because of the keyword `ordered` that makes sure that the statement considers triples `<c,t1,t2>` where `t1 < t2`. Of particular interest are the expressions `freq[<c,t1>]` and `freq[<c,t2>]` illustrating that the indices of array `freq` are records of the form `<c,t>`, where `c` is a cell and `t` is a transmitter. Note also that the distance is computed using the

```
nbCells = 25;
nbFreqs = 256;
nbTrans = [8 6 6 1 4 4 8 8 8 8 4 9 8 4 4 10 8 9 8 4 5 4 8 1 1];
distance = [
   [16 1 1 0 0 0 0 0 1 1 1 1 1 2 2 1 1 0 0 0 2 2 1 1 1]
   [1 16 2 0 0 0 0 0 2 2 1 1 1 2 2 1 1 0 0 0 0 0 0 0 0]
   [1 2 16 0 0 0 0 0 2 2 1 1 1 2 2 1 1 0 0 0 0 0 0 0 0]
   [0 0 0 16 2 2 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1]
   [0 0 0 2 16 2 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1]
   [0 0 0 2 2 16 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1]
   [0 0 0 0 0 0 16 2 0 0 1 1 1 0 0 1 1 1 1 2 0 0 0 1 1]
   [0 0 0 0 0 0 2 16 0 0 1 1 1 0 0 1 1 1 1 2 0 0 0 1 1]
   [1 2 2 0 0 0 0 0 16 2 2 2 2 2 2 1 1 1 1 1 1 1 0 1 1]
   [1 2 2 0 0 0 0 0 2 16 2 2 2 2 2 1 1 1 1 1 1 1 0 1 1]
   [1 1 1 0 0 0 1 1 2 2 16 2 2 2 2 2 2 1 1 2 1 1 0 1 1]
   [1 1 1 0 0 0 1 1 2 2 2 16 2 2 2 2 2 1 1 2 1 1 0 1 1]
   [1 1 1 0 0 0 1 1 2 2 2 2 16 2 2 2 2 1 1 2 1 1 0 1 1]
   [2 2 2 0 0 0 0 0 2 2 2 2 2 16 2 1 1 1 1 1 1 1 1 1 1]
   [2 2 2 0 0 0 0 0 2 2 2 2 2 2 16 1 1 1 1 1 1 1 1 1 1]
   [1 1 1 0 0 0 1 1 1 1 2 2 2 1 1 16 2 2 2 1 2 2 1 2 2]
   [1 1 1 0 0 0 1 1 1 1 2 2 2 1 1 2 16 2 2 1 2 2 1 2 2]
   [0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 16 2 2 1 1 0 2 2]
   [0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 16 2 1 1 0 2 2]
   [0 0 0 1 1 1 2 2 1 1 2 2 2 1 1 1 1 2 2 16 1 1 0 1 1]
   [2 0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 1 1 1 16 2 1 2 2]
   [2 0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 1 1 1 2 16 1 2 2]
   [1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 16 1 1]
   [1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 1 2 2 1 16 2]
   [1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 1 2 2 1 2 16]];
};
```

Fig. 2. Instance Data for the Frequency-Allocation Problem (`alloc.dat`).

function abs, which computes the absolute value of its argument (which may be an arbitrary integer expression).

The second set of constraints handles the distance constraints between transmitters from different cells. The instruction

```
forall(ordered c1, c2 in Cells : distance[c1,c2] > 0)
    forall(t1 in 1..nbTrans[c1] & t2 in 1..nbTrans[c2])
       abs(freq[<c1,t1>] - freq[<c2,t2>]) >= distance[c1,c2];
```

considers each pair of distinct cells whose distance must be greater than zero and each two transmitters in these cells, and states that the distance between the frequencies of these transmitters must be at least the distance specified in the matrix distance.

Another interesting part of this model is the search strategy. The basic structure is not surprising: OPL considers each transmitter and chooses a frequency nondeterministically. The interesting feature of the model is the heuristic. OPL chooses to generate a value for the transmitter with the smallest domain and, in case of ties, for the transmitter whose cell size is as small as possible. This multicriterion heuristic is expressed using a tuple <dsize(freq[t]),nbTrans[t.c]> to obtain

```
forall(t in Transmits ordered by increasing <dsize(freq[t]),nbTrans[t.c]>)
```

Each transmitter is associated with a tuple $< s, c >$, where s is the number of its possible frequencies and c is the number of transmitters in the cell to which the transmitter belongs. A transmitter with tuple $< s_1, c_1 >$ is preferred over a transmitter with tuple $< s_2, c_2 >$ if $s_1 < s_2$ or if $s_1 = s_2$ and $c_1 < c_2$.

Once a transmitter has been selected, OPL generates a frequency for it in a nondeterministic manner. Once again, the model specifies a heuristic for the ordering in which the frequencies must be tried. To reduce the number of frequencies, the model says to try first those values that were used most often in previous assignments. This heuristic is implemented using a nondeterministic tryall instruction with the order specified using the nbOccur function (nbOccur(i,a) denotes the number of occurrences of i in array a at a given step of the execution):

```
forall(t in Transmits ordered by increasing <dsize(freq[t]),nbTrans[t.c]>)
    tryall(f in Freqs ordered by decreasing nbOccur(f,freq))
       freq[t] = f;
```

This search procedure is typical of many constraint satisfaction problems and consists of using a first heuristic to dynamically choose which variable to instantiate next (variable choice) and a second heuristic to choose which value to assign nondeterministically to the selected variable (value choice). The forall instruction is of course deterministic, while the tryall instruction is nondeterministic: potentially all possible values are chosen for the selected variable. Note that, on the instance depicted in Statement 2, OPL returns a solution with 95 frequencies in about 3 seconds.

3 Sport Scheduling

This section considers the sport-scheduling problem described in [7, 10]. The problem 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.

A solution to this problem for 8 teams is shown in Figure 3. 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.

	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

Fig. 3. A Solution to the Sport-Scheduling Application with 8 Teams

The sport-scheduling problem is an interesting application for constraint programming. On the one hand, it is a standard benchmark (submitted by Bob Daniel) to the well-known MIP library and it is claimed in [7] that state-of-the-art MIP solvers cannot find a solution for 14 teams. The OPL models presented in this section are computationally much more efficient. On the other hand, the sport-scheduling application demonstrates fundamental features of constraint programming including global and symbolic constraints. In particular, the model makes heavy use of arc-consistency [6], a fundamental constraint satisfaction techniques from artificial intelligence.

The rest of this section is organized as follows. Section 3.1 presents an OPL model that solves the 14-teams problem in about 44 seconds. Section 3.2 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 [10].

3.1 A Simple OPL model

The simple model is depicted in Statement 4. Its input is the number of teams `nbTeams`. A number of 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 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 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;

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

search {
   generate(game);
};
```

Fig. 4. A Simple Model for the Sport-Scheduling Model.

```
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` 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 f; int s; 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 f,int s,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 in the model follow almost directly the problem description. The constraint

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

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 and an annotation `onDomain` to enforce arc consitency. See [8] for a description on how to enforce arc consistency on this global constraint. The constraint

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

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`. The annotation `extendedPropagation` specifies to enforce arc consistency on this constraint. See [9] for a description on how to enforce arc consistency on this global constraint. The constraint

```
alldifferent(game) onDomain;
```

specifies that all games are different, i.e., that all teams play against each other team. These constraints illustrate some of the global constraints of OPL. Other global constraints in the current version include a sequencing constraint, a circuit constraint, and a variety of scheduling constraints. 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-consitency on this symbolic constraint.

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 modern PC (400mhz).

3.2 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 Statements 5 and 6.

The main novelty in the statement is the array `roundRobin` that specifies the games for every week. Assuming that n denotes the number of teams, the basic idea is to fix the set of games of the first week as

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

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

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

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

Fig. 5. A Round-Robin Model for the Sport-Scheduling Model (Part I).

```
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]) onDomain;
   alldifferent(game) onDomain;
   forall(p in Periods)
      distribute(occur,values,all(w in EWeeks & s in Slots) team[p,w,s])
         extendedPropagation;
   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]);
   };
};
```

Fig. 6. A Round-Robin Model for the Sport-Scheduling Model (Part II).

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

Fig. 7. Experimental Results for the Sport-Scheduling Model

and

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

This round-robin schedule is computed in the `initialize` instruction and the last instruction 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 7 depicts the experimental results for various numbers of teams. It is possible to improve model further by exploiting even more symmetries: see [10] for complete details.

4 Job-Shop Scheduling

One of the other significant features of OPL is its support for scheduling applications. OPL has a variety of domain-specific concepts for these applications that are translated into state-of-the-art algorithms. To name only a few, they include the concepts of activities, unary, discrete, and state resources, reservoirs, and breaks as well as the global constraints linking them.

Statement 8 describes a simple job-shop scheduling model. The problem is to schedule a number of jobs on a set of machines to minimize completion time, often called the *makespan*. Each job is a sequence of tasks and each task requires a machine. Statement 8 first declares the number of machines, the number of jobs, and the number of tasks in the jobs. The main data of the problem, i.e., the duration of all the tasks and the resources they require, are then given. The next set of instructions

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

is most interesting. The first instruction describes the schedule horizon, i.e., the date by which the schedule should be completed at the lastest. In this application, the schedule horizon is given as the summation of all durations, which is clearly an upper bound on the duration of the schedule. The next instruction declares the activities of the problem. Activities are first-class objects in OPL and can be viewed (in a first approximation) as consisting of variables representing the starting date, the duration, and the end date of a task, as well as the constraints linking them. The variables of an activity are accessed as fields of records. In our application, there is an activity associated with each task of each job. The instruction

```
int nbMachines = ...;
range Machines 1..nbMachines;
int nbJobs = ...;
range Jobs 1..nbJobs;
int nbTasks = ...;
range Tasks 1..nbTasks;
Machines resource[Jobs,Tasks] = ...;
int+ duration[Jobs,Tasks] = ...;
int totalDuration = sum(j in Jobs, t in Tasks) duration[j,t];

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

minimize
   makespan.end
subject to {
   forall(j in Jobs)
      task[j,nbTasks] precedes makespan;
   forall(j in Jobs & t in 1..nbTasks-1)
      task[j,t] precedes task[j,t+1];
   forall(j in Jobs & t in Tasks)
      task[j,t] requires tool[resource[j,t]];
};

search {
   LDSearch() {
      forall(r in Machines ordered by increasing localSlack(tool[r]))
         rank(u[r]);
   }
}
```

Fig. 8. A Job-Shop Scheduling Model (`jobshop.mod`).

```
UnaryResource tool[Machines];
```

declares an array of unary resources. Unary resources are, once again, first-class objects of OPL; they represent resources that can be used by atmost one activity at anyone time. In other words, two activities using the same unary resource cannot overlap in time. Note that the makespan is modeled for simplicity as an activity of duration zero.

Consider now the problem constraints. The first set of constraints specifies that the activities associated with the problem tasks precede the makespan activity. The next two sets specify the precedence and resource constraints. The resource constraints specify which activities require which resource. Finally, the search procedure

```
search {
    LDSearch() {
       forall(r in Machines ordered by increasing localSlack(tool[r]))
          rank(u[r]);
    }
}
```

illustrates a typical search procedure for job-shop scheduling and the use of limited discrepancy search (LDS) [5] as a search strategy. The search procedure

```
forall(r in Machines ordered by increasing localSlack(tool[r]))
    rank(u[r]);
```

consists of ranking the unary resources, i.e., choosing in which order the activities execute on the resources. The instruction `LDSearch()` specifies that the search space specified by the search procedure defined above must be explored using limited discrepancy search. This strategy, which is effective for many scheduling problems, assumes the existence of a good heuristic. Its basic intuition is that the heuristic, when it fails, probably would have found a solution if it had made a small number of different decisions during the search. The choices where the search procedure does not follow the heuristic are called *discrepancies*. As a consequence, LDS systematically explores the search tree by increasing the number of allowed discrepancies. Initially, a small number of discrepancies is allowed. If the search is not successful or if an optimal solution is desired, the number of discrepancies is increased and the process is iterated until a solution is found or the whole search space has been explored. Note that, besides the default depth-first search and LDS, OPL also supports best-first search, interleaved depth-first search, and depth-bounded limited discrepancy search. It is interesting to mention that this simple model solves MT10 in about 40 seconds and MT20 in about 0.4 seconds.

5 A Configuration Problem

This section illustrates OPLScript, a script language for controlling and composing OPL models. It shows how to solve an application consisting of a sequence of two models: a constraint programming model and an integer program. The application is a configuration problem, known as Vellino's problem, that is a small but good representive of many similar applications. For instance, complex sport scheduling applications can be solved in a similar fashion.

Given a supply of components and bins of various types, Vellino's problem consists of assigning the components to the bins so that the bin constraints are satisfied and the smallest possible number of bins is used. There are five types of components, i.e., glass, plastic, steel, wood, and copper, and three types of bins, i.e., red, blue, green. The bins must obey a variety of configuration constraints. Containment constraints specify which components can go into which bins: red bins cannot contain plastic of steel, blue bins cannot contain wood or plastic, and green bins cannot contain steel or glass. Capacity constraints specify a limit for certain component types for some bins: red bins contain at most one wooden component and green bins contain at most two wooden components. Finally, requirement constraints specify some compatibility constraints between the components: wood requires plastic, glass excludes copper and copper excludes plastic. In addition, we are given an initial capacity for each bin, i.e., red bins have a capacity of 3 components, blue bins of 1 and green bins of 4 and a demand for each component, i.e., 1 glass, 2 plastic, 1 steel, 3 wood, and 2 copper components.

```
Model bin("genBin.mod","genBin.dat");
import enum Colors bin.Colors;
import enum Components bin.Components;
struct Bin { Colors c; int n[Components]; };
int nbBin := 0;
Open Bin bins[1..nbBin];
while bin.nextSolution() do {
   nbBin := nbBin + 1;
   bins.addh();
   bins[nbBin].c := bin.c;
   forall(c in Components)
      bins[nbBin].n[c] := bin.n[c];
}
Model pro("chooseBin.mod","chooseBin.dat");
if pro.solve() then
   cout << "Solution at cost: " << pro.objectiveValue() << endl;
```

Fig. 9. A Script to Solve Vellino's Problem (`vellino.osc`) .

The strategy to solve this problem consists of generating all the possible bin configurations and then to choose the smallest number of them that meet the demand. This strategy is implemented using a script `vellino.osc` depicted in

```
enum Colors ...;
enum Components ...;
int capacity[Colors] = ...;
int maxCapacity = max(c in Colors) capacity[c];
var Colors c;
var int n[Components] in 0..maxCapacity;
solve {
   0 < sum(c in Components) n[c] <= capacity[c];
   c = red => n[plastic] = 0 & n[steel] = 0 & n[wood] <= 1;
   c = blue => n[plastic] = 0 & n[wood] = 0;
   c = green => n[glass] = 0 & n[steel] = 0 & n[wood] <= 2;
   n[wood] >= 1 => n[plastic] >= 1;
   n[glass] = 0 \/ n[copper] = 0;
   n[copper] = 0 \/ n[plastic] = 0;
};
```

Fig. 10. Generating the Bins in Vellino's Problem (`genBin.mod`) .

```
import enum Colors;
import enum Components;
struct Bin { Colors c; int n[Components]; };
import int nbBin;
import Bin bin[1..nbBin];
range R 1..nbBin;
int demand[Components] = ...;
int maxDemand = max(c in Components) demand[c];
var int produce[R] in 0..maxDemand;
minimize
   sum(b in R) produce[b]
subject to
   forall(c in Components)
      sum(b in R) bin[b].n[c] * produce[b] = demand[c];
```

Fig. 11. Choosing the Bins in Vellino's Problem (`chooseBin.mod`) .

Figure 9 and two models genBin.mod and chooseBin.mod depicted in Figures 10 and 11. It is interesting to study the script in detail at this point. The instruction

```
Model bin("genBin.mod","genBin.dat");
```

declare the first model. Models are, of course, a fundamental concept of OPLSCRIPT: they support a variety of methods (e.g., solve and nextSolution), their data can be accessed as fields of records, and they can be passed as parameters to procedures. The instructions

```
import enum Colors bin.Colors;
import enum Components bin.Components;
```

import the enumerated types from the model to the script; these enumerated types will be imported by the second model as well. The instructions

```
struct Bin { Colors c; int n[Components]; };
int nbBin := 0;
Open Bin bins[1..nbBin];
```

declare a variable to store the number of bin configurations and an open array to store the bin configurations themselves. Open arrays are arrays that can grow and shrink dynamically during the execution. The instructions

```
while bin.nextSolution() do {
    nbBin := nbBin + 1;
    bins.addh();
    bins[nbBin].c := bin.c;
    forall(c in Components)
       bins[nbBin].n[c] := bin.n[c];
}
```

enumerate all the bin configurations and store them in the bin array in model pro. Instruction bin.nextSolution() returns the next solution (if any) of the model bin. Instruction bins.addh increases the size of the open array (addh stands for "add high"). The subsequent instructions access the model data and store them in the open array. Once this step is completed, the second model is executed and produces a solution at cost 8.

Model genBin.mod specifies how to generate the bin configurations: It is a typical constraint program using logical combinations of constraints that should not raise any difficulty. Model chooseBin.mod is an integer program that chooses and minimizes the number of bins. This model imports the enumerated types as mentioned previously. It also imports the bin configurations using the instructions

```
import int nbBin;
import Bin bin[1..nbBin];
```

It is important to stress to both models can be developed and tested independently since import declarations can be initialized in a data file when a model is run in isolation (i.e., not from a script). This makes the overall design compositional.

6 Conclusion

The purpose of this paper was to review, through four applications, a number of constraint programming features of OPL to give a basic understanding of the expressiveness of the language. These features include very high-level algebraic notations and data structures, a rich constraint programming language supporting logical, higher-level, and global constraints, support for scheduling and resource allocation problems, and search procedures and strategies. The paper also introduced briefly OPLSCRIPT, a script language to control and compose OPL models. The four applications presented in this paper should give a preliminary, although very incomplete, understanding of how OPL can decrease development time significantly.

References

1. J. Bisschop and A. Meeraus. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study*, 20:1–29, 1982.
2. A. Colmerauer. An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418, 1990.
3. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.
4. R. Fourer, D. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* The Scientific Press, San Francisco, CA, 1993.
5. W.D. Harvey and M.L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.
6. A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
7. K. McAloon, C. Tretkoff, and G. Wetzel. Sport League Scheduling. In *Proceedings of the 3th Ilog International Users Meeting*, Paris, France, 1997.
8. J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94, proceedings of the Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.
9. J-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI-96, proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 209–215, Portland, Oregon, 1996.
10. J-C. Régin. Sport league scheduling. In *INFORMS*, Montreal, Canada, 1998.
11. Ilog SA. Ilog Solver 4.31 Reference Manual, 1998.
12. G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today.* LNCS, No. 1000, Springer Verlag, 1995.
13. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.