

A Preview of OPL

Pascal Van Hentenryck

Department of Computing Science and Engineering
UCL
2, Place Sainte-Barbe, B-1348, Louvain-la-Neuve (Belgium)
Email: pvh@info.ucl.ac.be

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 using 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 strengths. 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 [15], and its development environment OPL STUDIO.

OPL is a modeling language 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, or C++ or COM components, to integrate an OPL model in a larger application. The code generation produces a class for each model object 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 functionalities of OPL and

OPLSCRIPT through a number of applications, including a transportation problem, sport and job-shop scheduling applications, and a configuration problems.

2 A Transportation Problem

This section illustrates some of the functionalities of OPL to solve large-scale mathematical programming problems. Consider, for instance, a transportation problem where products must be shipped from a set of cities to another set of cities. Each city may ship, or may request, some units of each product. The demand of the cities must be met and their supply must be shipped. In addition, there is a constraint specifying that the total shipment for all products transported between two cities may not exceed a specified limit. The goal is to minimize the transportation cost while satisfying the constraints.

Statement 1 shows a simple model for this problem, which implicitly assumes that all cities are connected and that all products may be shipped between two cities. The statement starts by declaring two enumerated types for the cities and the products. Their initializations are given in a separated data file to isolate the model from the instance data. It then specifies the capacity limit on the connections, the supply and demand for each product and each city, and the transportation cost of a product between two cities. The decision variables in this model are of the form `trans[p,o,d]` and denotes the quantity of product `p` shipped between cities `o` and `d`. The rest of the statement specifies a linear objective function and linear constraint. The objective function minimizes the transportation cost

```
sum(p in Products & o,d in Cities) cost[p,o,d]*trans[p,o,d]
```

and sums over all products and cities. Note also the constraint

```
forall(o, d in Cities)
    sum(p in Products) trans[p,o,d] <= limit;
```

which makes sure that the products sent between two cities do not exceed the capacity.

The above model is not appropriate for large-scale problems where only a fraction of the cities are connected. For instance, for 100 cities and 100 products, there are already 10000 ways of shipping a product between two cities, a fraction of which would probably be relevant in practice.

```
enum Cities ...;
enum Products ...;
float+ limit = ...;
float+ supply[Products,Cities] = ...;
float+ demand[Products,Cities] = ...;
float+ cost[Products,Cities,Cities] = ...;

var float+ trans[Products,Cities,Cities];
minimize
    sum(p in Products & o,d in Cities)
        cost[p,o,d] * trans[p,o,d]
subject to {
    forall(p in Products & o in Cities)
        sum(d in Cities) trans[p,o,d] = supply[p,o];
    forall(p in Products & d in Cities)
        sum(o in Cities) trans[p,o,d] = demand[p,d];
    forall(o, d in Cities)
        sum(p in Products) trans[p,o,d] <= limit;
};
```

Figure 1: A Simple Transportation Model (transp.mod).

To exploit sparsity often present in large-scale mathematical programming model, it is useful to reflect the structure of the application closely. Statement 2 depicts a model illustrating this principle. The model is based on two main ideas: the concept of a connection between two cities and the concept of a route, i.e., the association of a product and a connection. The data is represented by a set `routes` of records of type

```
struct Connection { Cities o; Cities d; };
struct Route { Connection e; Products p; };
```

The array `cost` and `trans` can then be indexed with this set illustrating one of the appealing features of OPL: the ability to index an array by an arbitrary data. The data for the supplies and demands are also represented in a sparse way by projecting the set `routes` to obtain their index sets. In addition to that, the model also pre-computes, in a generic way, the cities `orig[p]` that can ship product `p` and the cities `dest[p]` that can receive product `p`, as well as some other information. The rest of the model is generally similar to the non-sparse model but reflects the new data organization. Note the simplicity of the objective function

```
minimize
    sum(r in Routes) cost[r] * trans[r]
```

which sums directly over the routes and the instruction

```
forall(c in connections)
    sum(<c,p> in routes) trans[<c,p>] <= limit;
```

which generates the capacity constraints efficiently. First, it iterates over the routes, not over all pairs of cities. Second, the aggregate operator `sum` uses parameter `c` to index the set `routes`, retrieving the relevant products effectively.

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:

```

enum Cities ...;
enum Products ...;

struct Connection { Cities o; Cities d; };
struct Route { Connection e; Products p; };
struct Supplier { Products p; Cities o; };
struct Customer { Products p; Cities d; };

{Route} Routes = ...;
{Connection} Connections = { c | <c,p> in Routes };
{Supplier} Suppliers = { <p,c.o> | <c,p> in Routes };
float+ supply[Suppliers] = ...;
{Customer} Customers = { <p,c.d> | <c,p> in Routes };
float+ demand[Customers] = ...;
float+ lim = ...;
float+ cost[Routes] = ...;
{Cities} orig[p in Products] = { c.o | <c,p> in Routes };
{Cities} dest[p in Products] = { c.d | <c,p> in Routes };
{Connection} CP[p in Products] = { c | <p,c> in Routes };

var float+ trans[Routes];

minimize
  sum(r in Routes) cost[r] * trans[r]
subject to {
  forall(p in Products & o in orig[p])
    sum(<o,d> in CP[p]) trans[<<o,d>,p>] = supply[<p,o>];
  forall(p in Products & d in dest[p])
    sum(<o,d> in CP[p]) trans[<<o,d>,p>] = demand[<p,d>];
  forall(c in Connections)
    sum(<c,p> in Routes) trans[<c,p>] <= lim;
};

```

Figure 2: A Sparse Transportation Model (stransp.mod).

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

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.

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 presents a simple OPL model that solves the 14-teams problem in about 44 seconds. See [14] for an even more efficient model. Both models are based on the constraint programs presented in [10].

The simple model is depicted in Figure 4. 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 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];

predicate link(int f,int s,int g)
    return g = (f-1) * nbTeams + s;

solve {
    forall(w in EWeeks)
        alldifferent(
            all(p in Periods & s in Slots) team[p,w,s]);
    alldifferent(game) onDomain;
    forall(p in Periods)
        distribute(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 4: A Simple Model for the Sport-Scheduling Model.

```
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 thought 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 instruction

```
predicate link(int f,int s,int g)  
  return g = (f-1) * nbTeams + s;
```

defines a predicate constraint which links the home and away teams with the identifier of the game. The predicate is used subsequently to state constraints that are made arc-consistent by the OPL implementation.

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] );
```

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 OPL enforces arc consistency on this constraint. 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] );
```

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 . Once again, OPL enforces arc consistency on this constraint. See [9] for a description on how to enforce arc consistency on this global constraint. The constraint

```
alldifferent(game);
```

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-consistency 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).

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.

Figure 5 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. Figure 5 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

```

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(tool[r]);
        }
    }
}

```

Figure 5: A Job-Shop Scheduling Model (jobshop.mod).

```

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 latest. 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

```
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 at most 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(tool[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. Once the resources are ranked, it is easy to find a solution. The procedure ranks first the resource with the smallest local slack (i.e., the machine that seems to be the most difficult to schedule) and then considers the remaining resource using a similar heuristic. 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, which is a small but good representative 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 or steel, blue bins cannot contain wood or

```

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($!t) then{at cost: ";
    cout << pro.objectiveValue() << endl;
}

```

Figure 6: A Script to Solve Vellino's Problem (`vellino.osc`).

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.

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 the script depicted in Figure 6 and two models `genBin.mod` and `chooseBin.mod` depicted in Figures 7 and 8. It is interest-

```
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;
};
```

Figure 7: 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 bins[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) bins[b].n[c]*produce[b] = demand[c];
```

Figure 8: Choosing the Bins in Vellino's Problem (chooseBin.mod).

ing 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 bins[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 features of OPL to give a preliminary 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 Twelfth 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.
- [14] P. Van Hentenryck, L. Michel, L. Perron, and J.C. Regin”, Constraint Programming in OPL. in *Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP’99)*, Paris, France, 1999
- [15] P. Van Hentenryck. *OPL Script: Composing and Controlling Models*. *Proceedings of the 1999 Workshop of the ERCIM Working Group on Constraints*, Paphos, Cyprus, 1999