OPL++: A Modeling Layer for Constraint Programming Libraries

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-00-07
December 2000

OPL++: A Modeling Layer for Constraint Programming Libraries

Laurent Michel and Pascal Van Hentenryck
Brown University
Box 1910
Providence, RI 02912
Email: pvh@cs.brown.edu

Abstract

Mathematical modeling and constraint programming languages have orthogonal strengths in stating combinatorial optimization problems. Modeling languages typically feature high-level set and algebraic notations, while constraint programming languages provide a rich constraint language and the ability to specify search procedures. This paper shows that many of the functionalities typically found in modeling languages can be integrated elegantly in constraint programming libraries without defining a specific language or preprocessor. In particular, it presents the design of OPL++, a C++ modeling layer for constraint programming that combines the salient features of both approaches. Of particular interest is the one-to-one correspondance between high-level models and OPL++ statements and the negligible overhead induced by the extensions.

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 optimization tools in order to simplify the design and implementation of combinatorial optimization problems. Their goal is to decrease development time substantially while preserving most of the efficiency of specialized programs. Most tools can be classified in two categories: mathematical modeling languages and constraint programming languages. Modeling languages such as AMPL [9] and GAMS [2] provide high-level algebraic and set notations to express, in concise ways, 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 [6], Prolog III [4], Eclipse [7], Prolog IV [5], Oz [15, 16], Salsa [12], and Ilog Solver [11] have orthogonal strengths. Their constraint vocabulary and their underlying solvers go beyond traditional linear and nonlinear constraints and support logical, higher-order, and global constraints. They also make it possible to program search procedures to specify how to explore the search space. However, these languages are mostly aimed at computer scientists and often have weaker abstractions for algebraic and set manipulation.

The orthogonal strengths of mathematical modeling and constraint programming languages were a primary motivation behind the optimization programming language OPL [17], a modeling language which unifies modeling and constraint programming languages. OPL shares with modeling

languages their high-level algebraic and set notations. It also contains some novel functionalities to exploit sparsity in large-scale applications, such as the ability to index arrays with arbitrary data structures. OPL shares with constraint programming languages their rich constraint vocabulary, their support for scheduling and resource allocation problems, and the ability to specify search procedures and strategies. Note also that a language like Salsa [12] shares some of these motivations and, on some aspects, offers similar functionalities.

The purpose of this paper is to show that most of the functionalities of modeling languages can be supported elegantly in constraint programming libraries. In particular, it presents the design of OPL++, a C++ modeling layer for constraint programming that combines the salient features of both approaches. In addition to the traditional functionalities of constraint programming[1], OPL++ supports a variety of novel features typically found in modeling or domain-specific languages, including high-level set, algebraic, logical, and array operators, multi-dimensional arrays, sparse data structures, and high-level constructs for specifying search procedures. Of particular interest is the one-to-one correspondance between many high-level models (e.g., OPL models) and OPL++ statements and the negligible overhead induced by the extensions.

OPL++ offers a number of significant benefits. First, it makes available, inside traditional object-oriented languages, the advanced features typically found in mathematical modeling languages and significantly advances the modeling functionalities of existing libraries. As a consequence, it leads to higher-level programs that are easier to implement and to maintain. Second, it simplifies the implementation of modeling languages, since the distance between a model and the corresponding library code is now substantially reduced. Third, OPL++ significantly eases the integration of modeling languages in larger applications. Indeed, recent modeling languages, such as XPRESS-MP and OPL, make it possible to integrate a model inside an application through code generation or through COM or C++ components (e.g., [18]). Since the modeling language and the library now share the same control and data structures, their integration is greatly simplified. Note that OPL++ does not preclude the need for modeling languages which target a different audience and have more freedom with respect to the syntax and the type system of the language. Finally, although only constraint programming is considered in this paper, the functionalities of OPL++ apply equally well to mathematical programming and local search libraries. Companion papers will discuss these aspects of OPL++ separately.

The rest of this paper illustrates OPL++ through a number of traditional constraint programming applications, starting with simple examples such as the magic series problem and concluding with more realistic applications such as frequency allocation and sport scheduling. The main focus of the paper is to show the similarity between OPL models and OPL++ statements and to give some experimental results to show that these functionalities do not prevent OPL++ to be competitive with state-of-the-art constraint programming systems [8].

2 Magic Series

The magic-series problem has become a traditional constraint-programming benchmark. The problem consists of finding a magic series, i.e., a sequence of numbers $S = (s_0, s_1, \ldots, s_n)$ such that s_i

[1] OPL++ is built on top of the novel open constraint programming library Solver++.

```
void magic(int n)
{
   OplCPManager mgr;
   OplIntRange Dom(mgr,0,n-1);
   OplIntVarArray s(mgr,Dom,Dom,"s");
   OplIntParameter i(mgr), j(mgr);

   mgr.post(OplForall(i,Dom,s[i] == OplSum(j,Dom,s[j]==i)));
   mgr.post(OplSum(j,Dom,s[j]*j)==n);

   if (mgr.newSearch().search(OplGenerate(s)))
      cout << s << endl;
}
```

Figure 1: Magic Series in OPL++.

represents the number of occurrences of i in S. For instance, $(1, 2, 1, 0)$ is a magic series of size 4, since there is one occurrence of 0, two occurrences of 1, one occurrence of 1 and zero occurrences of 3. Figure 1 depicts a solution to the magic series problem. It introduces the use of parameters and aggregate operators in OPL++. The instruction

```
OplIntParameter i(mgr), j(mgr);
```

declares two integer parameters `i` and `j` to be used in aggregate operators. The instruction

```
mgr.post(OplForall(i,Dom,s[i] == OplSum(j,Dom,s[j]==i));
```

states all cardinality constraints using a `OplForall` operator. It states constraints of the form

```
s[i] == OplSum(j,Dom,s[j]==i)
```

where `OplSum(j,Dom,s[j]==i)` counts the occurrences of `i` in `s`. The instruction

```
mgr.post(OplSum(j,Dom,s[j]*j)==n);
```

reuses parameter `j` to state the redundant constraint typically found in this problem. Finally, the instruction

```
mgr.newSearch().search(OplGenerate(s))
```

```
void magic(int n)
{
   OplCPManager mgr;
   OplIntRange Dom(mgr,0,n-1);
   OplIntVarArray s(mgr,Dom,Dom,"s");
   OplIntParameter j(mgr);

   mgr.post(OplExactly(s,s,OplBasicPropagation));
   mgr.post(OplSum(j,Dom,s[j]*j)==n);

   if (mgr.newSearch().search(OplGenerate(s)))
      cout << s << endl;
}
```

Figure 2: Magic Series with a Global Constraint in OPL++.

	25	50	100	200	400	800	1600
CPU Time (sec.)	0.01	0.01	0.03	0.10	0.37	1.45	6.20

Table 1: Computation Results for the Magic Series Problem

explores the search tree specified by `OplGenerate(s)` which assigns values to variables using the first-fail principle. *Observe the similarity between the* OPL++ *and* OPL *statements* [17]: the main difference is the explicit declaration of the parameters needed in C++. Figure 2 depicts a similar statement which uses a global constraint `OplExactly` to specify the cardinality constraint. Table 1 depicts the efficiency result of this last statement on a 300 Mhz PC running Linux. It shows that, on this standard benchmark [8], OPL++ is at least as efficient as the best (commercial and academic) systems, showing that its additional functionalities do not introduce any significant overhead.

3 Stable Marriages

The stable marriage problem (e.g., [10]) is a beautiful example to illustrate a number of features of OPL++. In particular, it shows that OPL++ supports multidimensional arrays which can be indexed by expressions containing variables. The problem can be described as follows. Consider a group of women and a group of men who must marry and assume that each person has indicated a ranking for her/his possible spouses. The problem is to find a matching between the two groups such that the marriages are stable. The definition of stability is interesting: a marriage between m and w is

```
void stableMarriage(OplCPManager mgr, OplIntRange Men,OplIntRange Women,
                    OplIntMatrix<2> rankMen, OplIntMatrix<2> rankWomen)
{
   OplIntVarArray wife(mgr,Men,Women,"wife");
   OplIntVarArray husband(mgr,Women,Men,"husband");
   OplIntParameter m(mgr), w(mgr);

   mgr.post(OplForall(m,Men,husband[wife[m]]==m));
   mgr.post(OplForall(w,Women,wife[husband[w]]==w));

   mgr.post(OplForall(m,Men,
               OplForall(w,Women,
                  OplImply(rankMen[m][w] < rankMen[m][wife[m]],
                           rankWomen[w][husband[w]] < rankWomen[w][m]))));
   mgr.post(OplForall(w,Women,
               OplForall(m,Men,
                  OplImply(rankWomen[w][m] < rankWomen[w][husband[w]],
                           rankMen[m][wife[m]] < rankMen[m][w]))));
   if (mgr.newSearch().search(OplGenerate(wife)))
               cout << wife << endl;
}
```

Figure 3: Stable Marriages in OPL++.

stable provided that whenever m prefers another person o to w, o prefers her/his spouse to m and, vice-versa, whenever w prefers another person o to m, o prefers her/his spouse to w. Intuitively, m and w may be unhappy but they are bound to stay together.

Figure 3 describes the core function of this application. It receives as parameters two ranges representing the sets of men and women as well as two matrices (of dimension 2) describing the rankings (of the women by the men and vice-versa). The model is expressed in terms of two arrays of variables: `wife`, which specifies the men's wives, and `husband`, which specifies the women's husbands. Variables in array `wife` must take their values in set `Women`, while variables in array `husband` take their values in set `Men`. The constraints in this problem are of course the interesting and novel part. The first two sets of constraints

```
OplForall(m,Men,husband[wife[m]]==m)
OplForall(w,Women,wife[husband[w]]==w)
```

guarantee that a solution is a set of marriages by stating the spouse of the spouse of a person is the person herself. OPL++ enforces arc consistency on these constraints which feature expressions

where arrays of variables are indexed by variables. More interesting are the stability rules. The set of constraints

```
OplForall(m,Men,
   OplForall(w,Women,
      OplImply(rankMen[m][w] < rankMen[m][wife[m]],
               rankWomen[w][husband[w]] < rankWomen[w][m])))
```

states that, if a man `m` prefers a woman `w` to his wife (i.e., the rank of `w` is less than the rank of his wife), then `w` prefers her husband to `m`. These constraints index the two-dimensional array `rankMen` with variables `wife[m]`. *Once again, there is a one-to-one mapping with the corresponding* `OPL` *statement* [17].

4 The Queens Problem

The traditional queens problem now illustrates how search procedures can be specified at a very high level in `OPL++`. The solution described here uses redundant modeling (e.g., [3]) and models the problem using both a column and a row viewpoint as well as constraints to link them together. More specifically, the program uses a two-dimensional array of variables: `queen[0][c]` represents the row of the queen in column `c`, while `queen[1][r]` represents the column of the queen in row `r`.

Figure 4 depicts the `OPL++` program. It first declares a variety of parameters and the two-dimensional array of variables. The traditional constraints are then stated, both for the column and the row viewpoints. The constraints

```
OplForall(v,View,
   OplForall(OplOrdered(i,j),Dom,
      queen[v][i] != queen[v][j] &&
      queen[v][i] != queen[v][j]+i-j &&
      queen[v][i] != queen[v][j]+j-i))
```

features the `OplOrdered` construct in the `OplForAll` operator to ensure that `i` and `j` both takes their values in `Dom` but satify the condition `i < j`. The constraints

```
OplForall(v,View,
   OplForall(i,Dom,
      OplEquivalence(queen[0][i] == v,queen[1][v]== i)))
```

link the viewpoints together by specifying that, whenever the queen on column `i` is assigned row `v`, the queen on row `v` must be assigned column `i` (and vice-versa). Most interesting is the search procedure

```
OplAtom labeling =
    OplForall(q,Dom,OplIncreasing(queen[0][q]->getSize(),OplAbs(q-mid)),
              OplTryall(val,Dom,OplIncreasing(queen[1][val]->getSize()),
                        queen[0][q] == val));
```

```
void queens(int n)
{
   OplCPManager mgr;
   int mid = n/2;
   OplIntParameter i(mgr), j(mgr), v(mgr), q(mgr), val(mgr);
   OplIntRange View(mgr,0,1);
   OplIntRange Dom(mgr,1,n);
   OplIntVarMatrix<2> queen(mgr,View,Dom,Dom,"queen");

   mgr.post(OplForall(v,View,
               OplForall(OplOrdered(i,j),Dom,
                  queen[v][i] != queen[v][j] &&
                  queen[v][i] != queen[v][j]+i-j &&
                  queen[v][i] != queen[v][j]+j-i)));
   mgr.post(OplForall(v,View,
               OplForall(i,Dom,
                  OplEquivalence(queen[0][i] == v,queen[1][v]== i));
   OplAtom labeling =
      OplForall(q,Dom,OplIncreasing(queen[0][q]->getSize(),OplAbs(q-mid)),
                OplTryall(val,Dom,OplIncreasing(queen[1][val]->getSize()),
                          queen[0][q] == val));

   if (mgr.newSearch().search(labeling))
      cout << queen << endl;
}
```

Figure 4: A Queens Program using Redundant Modeling in OPL++.

```
if (mgr.newSearch().search(labeling))
    cout << queen << endl;
```

which features both a dynamic variable ordering and a dynamic value ordering. Intuitively, the `OplForall` instruction considers the column viewpoint and selects first the queen with the smallest domain and, in case of ties, the queen closest to the middle of the chessboard. Note the use of a lexicographic criterion to choose the queen to be placed. The `OplTryall` instruction chooses which value to assign to the selected queen. It selects first the row that corresponds to the queen with the smallest domain using the row viewpoint.

Observe the conciseness of the search procedure and the one-to-one mapping between the `OPL++` and the `OPL` search procedure. High-level specifications of search procedures have become integral part of modeling or domain-specific languages for constraint programming (e.g., [12, 17, 1]): *the novelty here is that these functionalities are now supported in* `OPL++` *without extending the host language (using preprocessors) or proposing a new language.*

5 Frequency Allocation

The frequency-allocation problem (e.g., [17]) illustrates a number of interesting features of `OPL++`, including the use of *sparse arrays*, a fundamental feature typically found in modeling languages, and the use of *tuple parameters*. The 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.

Figure 5 shows the core function of the `OPL++` program for the frequency-allocation problem which is a direct translation of the `OPL` model. It receives as input two ranges specifying the set of cells and of frequencies, an array `nbTrans` specifying the number of transmitters in each cell, and a 2-dimensional matrix specifying the distance between cells. The first interesting feature of this model is the computation of the set of transmitters. The instruction

```
OplTupleSet Transmitters(OplSetCollect(c,Cells,i,OplRange(1,nbTrans[c]),OplTuple(c,i)));
```

specifies that `Transmitters` is a set of tuples of the form `<c,i>` where `c` is a cell and `i` is the transmitter number in that cell, i.e., a number between 1 and `nbTrans[c]`. This set is computed automatically using the `OplSetCollect` construct. The second interesting feature of the model is how variables are declared. The instruction

```
void frequencyAllocation(OplCPManager mgr,OplIntRange Cells,OplIntRange Freqs,
                         OplIntArray nbTrans,OplIntMatrix<2> dist)
{
   OplIntParameter c(mgr), t1(mgr), t2(mgr), c1(mgr), c2(mgr), i(mgr), f(mgr);
   OplTupleParameter t(mgr);

   OplTupleSet Transmitters(
      OplSetCollect(c,Cells,i,OplRange(1,nbTrans[c])),OplTuple(c,i)));
   OplIntVarSparseArray freq(mgr,Transmitters,Freqs,"freq");
   OplRevIntArray occ(mgr,Freqs,"occ");

   mgr.post(OplOccurence(freq,occ));
   mgr.post(OplForall(c,Cells,
               OplForall(OplOrdered(t1,t2),OplRange(1,nbTrans[c]),
                  OplAbs(freq[OplTuple(c,t1)]-freq[OplTuple(c,t2)]) >= 16)));
   mgr.post(
      OplForall(OplOrdered(c1,c2),Cells,
         OplForall(t1,OplRange(1,nbTrans[c1]),
            OplForall(t2,OplRange(1,nbTrans[c2]),
               OplAbs(freq[OplTuple(c1,t1)]-freq[OplTuple(c2,t2)]) >= dist[c1][c2]))));
   OplAtom labeling =
      OplForall(t,Transmitters,OplIncreasing(freq[t]->getSize(),nbTrans[t->get(0)]),
                OplTryall(f,Freqs,OplDecreasing(occ[f]),
                          freq[t]==f));
   if (mgr.newSearch().search(labeling))
      cout << freq << endl;
}
```

Figure 5: Frequency Allocation in OPL++.

```
OplIntVarSparseArray freq(mgr,Transmitters,Freqs,"freq");
```

declares an array of variables indexed by elements of `Transmitters`, i.e., by tuples. These variables represent the frequencies assigned to the transmitters. Observe that `OPL++` allows arrays to be indexed by any `OPL++` data structure, a functionality often instrumental in obtaining more natural and efficient statements in modeling languages. Consider now the instruction

```
OplForall(c,Cells,
   OplForall(OplOrdered(t1,t2),OplRange(mgr,1,nbTrans[c]),
      OplAbs(freq[OplTuple(c,t1)]-freq[OplTuple(c,t2)]) >= 16))
```

which states the distance constraints between two transmitters within a cell. Note how the array `freq` is indexed by tuples composed of a cell and a transmitter number. The final interesting part of this model is the search strategy:

```
OplForall(t,Transmitters,OplIncreasing(freq[t]->getSize(),nbTrans[t->get(0)]),
          OplTryall(f,Freqs,OplDecreasing(occ[f]),
                    freq[t]==f))
```

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. 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 `tryall` instruction with the order specified using the `occ` array, which maintains the number of occurrences of each frequency in the current instantiation. Of particular interest here is the `OplForall` instruction which iterates over a set of tuples using the tuple parameter `f` as well the expressions `freq[f]` and `occ[f]` which index arrays with the tuple parameter. Once again, these new features make the model very close to the corresponding `OPL` model.

6 Sport Scheduling

This section describes a constraint programming solution to the sport-scheduling problem, a well-known benchmark in mathematical programming submitted by Bob Daniel to the well-known MIP library. It illustrates several interesting features of `OPL++`, resulting in a very concise model. Only a simple solution which solves the 14 teams quickly is considered here. More elaborate models exist and a comprehensive discussion of this problem can be found in [14, 19]. 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:

	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 6: 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 6. 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 [13] 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 [13].

Figure 7 present a function to solve the sport-scheduling problem using the basic ideas in [14, 19]. Its input is the number of teams `nbTeams`. Several ranges are defined from the input: the teams, the weeks, the extended weeks, i.e., the weeks plus a dummy week, the periods, the slots which identify the home (0) and the away (1) teams, and the games.

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 every other team. Games are uniquely identified by their two teams and the set `setGames` establishes the link between two teams and their corresponding game. More precisely, the instruction

```
OplTupleSet setGames(OplSetCollect(OplOrdered(i,j),Teams,OplTuple(i,j,(i-1)*n+j-1)));
```

computes a set of tuples of the form `<i,j,nb>` where `i` and `j` are respectively the home and away teams of game `nb`. For 8 teams, this set consists of tuples of the form

```
<1,2,1>
<1,3,2>
...
<7,8,55>
```

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

```
void sport(int n)
{
   int np = n/2;
   OplCPManager mgr;
   OplIntParameter i(mgr), j(mgr), p(mgr), w(mgr), s(mgr);
   OplIntRange Teams(mgr,1,n);
   OplIntRange Weeks(mgr,1,n-1);
   OplIntRange EWeeks(mgr,1,n);
   OplIntRange Periods(mgr,1,np);
   OplIntRange Slots(mgr,0,1);
   OplIntRange Games(mgr,1,(n-1)*n);

   OplTupleSet setGames(
      OplSetCollect(OplOrdered(i,j),Teams,OplTuple(i,j,(i-1)*n+j-1)));
   OplIntVarMatrix<3> teams(mgr,Periods,EWeeks,Slots,Teams,"teams");
   OplIntVarMatrix<2> games(mgr,Periods,Weeks,Games,"games");

   mgr.post(OplForall(w,EWeeks,
               OplAlldifferent(OplCollect(p,Periods,s,Slots,teams[p][w][s]))));
   mgr.post(OplForall(p,Periods,
               OplExactly(2,Teams,OplCollect(w,EWeeks,s,Slots,teams[p][w][s]))));
   mgr.post(OplAlldifferent(games));
   mgr.post(OplForall(p,Periods,OplForall(w,Weeks,
               OplElementOf(teams[p][w][0],teams[p][w][1],games[p][w],setGames));
   mgr.post(OplForall(p,Periods,OplForall(w,EWeeks,
               teams[p][w][0] < teams[p][w][1]);

   if (mgr.newSearch().search(OplAnd(OplGenerate(games),OplGenerate(teams))))
      cout << teams << endl;
}
```

Figure 7: A Simple Model for the Sport-Scheduling Model in OPL++.

```
OplIntVarMatrix<3> teams(mgr,Periods,EWeeks,Slots,Teams,"teams");
OplIntVarMatrix<2> games(mgr,Periods,Weeks,Games,"games");
```

declares the two array of variables. Observe that `teams` is a 3-dimensional array. The constraint declarations follow almost directly from the problem description. The constraint

```
OplAlldifferent(OplCollect(p,Periods,s,Slots,teams[p][w][s]))
```

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

```
OplExactly(2,Teams,OplCollect(w,EWeeks,s,Slots,teams[p][w][s]))
```

specifies that a team plays exactly twice over the course of the "extended" season. It specifies that the values in `Teams` (second argument) must occur twice (first argument) in teams playing in period `p` (third argument). `OplExactly` is a special case of the cardinality constraint, another well-known global constraint. Once again, the `OPL++` implementation achieves arc consistency on this constraint. The constraint

```
OplAlldifferent(games)
```

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

```
OplElementOf(teams[p][w][0],teams[p][w][1],games[p][w],setGames)
```

is most interesting. It specifies that the game `game[p][w]` consists of the teams `team[p][w][0]` and `team[p][w][1]`. More precisely, it ensures that the tuple

```
<teams[p][w][0],teams[p][w][1],games[p][w]>
```

belongs to the set `setGames`, effectively linking the team and the game variables. The `OPL++` implementation enforces arc consistency on this symbolic constraint as well. The search procedure in this statement is extremely simple and consists of generating first values for the games. Note also that generating values for the games automatically assigns values to the teams by constraint propagation, except for the teams in the dummy week.

This model finds a solution for 14 teams in about 53 seconds on a 300mhz PC running Linux. Note that this is about 20% faster than the same model in the best constraint programming systems featured in [8]. As mentioned, more sophisticated models [14, 19] can improve efficiency significantly.

7 Conclusion

This paper presented the design of `OPL++`, a `C++` modeling layer for constraint programming that combines the strengths of mathematical modeling languages and constraint programming libraries. In addition to the traditional features of constraint programming, `OPL++` supports a variety of novel features typically found in modeling or domain-specific languages, including high-level set, algebraic, logical, and array operators, multi-dimensional arrays, sparse data structures, and high-level constructs for specifying search procedures. This paper illustrated, on some representative applications, that there is a one-to-one mappings between `OPL` models and the corresponding `OPL++` statements. As a consequence, `OPL++` leads to higher-level constraint programs that are easier to implement and to maintain. It also simplifies the implementation of modeling languages and significantly eases their integration in larger applications (e.g., through code generation or through `COM` or `C++` components [18]), since the modeling language and the library now share the same control and data structures. Experimental results indicate that the additional functionalities do not introduce any significant overhead.

References

[1] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-O: An Imperative Language that Supports Declarative Programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, September 1998.

[2] 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.

[3] B.M.W. Cheng, J.H.M. Lee, H.F. Leung, and Y.W. Leung. Speeding up Constraint Propagation by Redundant Modeling. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, Cambridge, MA, August 1996. Springer Verlag.

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

[5] A. Colmerauer. Spécification de Prolog IV. Technical report, Laboratoire d'informatique de Marseille, 1996.

[6] 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.

[7] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4), October 2000.

[8] A. Fernadez and P.M. Hill. A Comparative Study of Eight Constraint Programming Languages over the Boolean and Finite Domains. *Constraints*, 5(3), July 2000.

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

[10] D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms.* The MIT Press, Cambridge, MA, 1989.

[11] Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.

[12] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[13] K. McAloon, C. Tretkoff, and G. Wetzel. Sport League Scheduling. In *Proceedings of the 3th Ilog International Users Meeting*, Paris, France, 1997.

[14] J-C. Régin. Sport league scheduling. In *INFORMS*, Montreal, Canada, 1998.

[15] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[16] G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today.* LNCS, No. 1000, Springer Verlag, 1995.

[17] P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.

[18] P. Van Hentenryck, L. Michel, P. Laborie, W. Nuijten, and J. Rogerie. Combinatorial Optimization in OPL Studio. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence International Conference (EPIA'99)*, Evora, Portugal, September 1999. (Invited Paper).

[19] 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, September 1999. (Invited Paper).