Combinatorial Optimization in OPL Studio

P. Van Hentenryck[1], L. Michel[2], P. Laborie[2], W. Nuijten[2], and J. Rogerie[2]

[1] UCL, Place Sainte-Barbe, 2, B-1348 Louvain-La-Neuve, Belgium
Email: pvh@info.ucl.ac.be
[2] Ilog SA, 9 rue de Verdun, F-94253 Gentilly Cedex, France
Email: {ldm,laborie,nuijten,rogerie}@ilog.fr

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 are 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. Finally, OPL models can be embedded in larger application through C++ code generation. This paper presents an overview of these functionalities on a scheduling application.

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 [5] and GAMS [1] provide 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 [4], Prolog III and its successors [3], OZ [8], and Ilog Solver [7] 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

P. Barahona and J.J. Alferes (Eds.): EPIA'99, LNAI 1695, pp. 1–15, 1999.

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 [10], its associated script language OPLScript [9], 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 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 how to solve combinatorial optimization problems in OPL Studio using a scheduling application. It is of course impossible to cover even a reasonable fraction of the features available in OPL and OPLScript but the hope is to convey a flavor of these languages and an overview of the overall approach. See [11] for a companion paper describing the constraint programming features of OPL. The rest of this paper is organized as follows. Section 2 describes the OPL model for the scheduling applications, Section 3 how OPLScript can control the models, while Section 4 describes C++ code generation. All the models/scripts/programs can be run in ILOG OPL Studio 2.1.

2 The Modeling Language OPL

This section illustrates OPL on a scheduling application. The application demontrates various modeling concepts of OPL as well as some of the OPL support for scheduling applications. In particular, the application illustrates the concepts

of activities, unary resources, discrete resources, state resources, and transition times, giving a preliminary understanding of the rich support of OPL in this important area. To ease understanding, the application is presented in stepwise refinements starting with a simplified version of the problem and adding more sophisticated concepts incrementally.

2.1 The Basic Model

Consider an application where a number of jobs must be performed in a shop equipped with a number of machines. Each job corresponds to the processing of an item that needs to be sequentially processed on a number of machines for some known duration. Each item is initially available in area A of the shop. It must be brought to the specified machines with a trolley. After it has been processed on all machines, it must be stored in area S of the shop. Moving an item from area x to area y consists of (1) loading the item on the trolley at area x; (2) moving the trolley from area x to area y and (3) unloading the item from the trolley at area y. The goal is to find a schedule minimizing the makespan. In this version of the problem, we ignore the time to move the trolley and we assume that the trolley has unlimited capacity. Subsequent sections will remove these limitations.

The specific instance considered here consists of 6 jobs, each of which requires processing on two specified machines. As a consequence, a job consists of 8 tasks

1. load the item on the trolley at area A;
2. unload the item from the trolley at the area of the first machine required by the job;
3. process the item on the first machine;
4. load the item on the trolley at the area of this machine;
5. unload the item from the trolley at the area of the second machine required by the job;
6. process the item on the second machine;
7. load the item on the trolley at the area of this machine;
8. unload the item from the trolley at Area S;

Figures 1 and 2 depict an OPL model for this problem, while Figure 3 describes the instance data. This separation between model and data is an important feature of modeling languages.

The statement starts by defining the set of jobs, the set of tasks to be performed by the jobs, and the possible locations of the trolley. As can be seen in the instance data (see Figure 3), the tasks correspond to the description given previously. The trolley has five possible locations, one for each available machine, one for the arrival area, and one for the storage area. The statement then defines the machines and the data for the jobs, i.e., it specifies the two machines required for each job and the duration of the activities to be performed on these machines. The machines are identified by their locations for simplicity. The statement also specifies the duration of a loading task, which concludes the description of the input data.

```
enum Jobs ...;
enum Tasks ...;
enum Location ...;
{Location} Machines = ...;
struct jobRecord {
   Location machine1;
   int durations1;
   Location machine2;
   int durations2;
};
jobRecord job[Jobs] = ...;
int loadDuration = ...;

Location location[Jobs,Tasks];
initialize {
   forall(j in Jobs) {
      location[j,loadA] = areaA;
      location[j,unload1] = job[j].machine1;
      location[j,process1] = job[j].machine1;
      location[j,load1] = job[j].machine1;
      location[j,unload2] = job[j].machine2;
      location[j,process2] = job[j].machine2;
      location[j,load2] = job[j].machine2;
      location[j,unloadS] = areaS;
   };
};

int duration[Jobs,Tasks];
initialize {
   forall(j in Jobs) {
      duration[j,loadA] = loadDuration;
      duration[j,unload1] = loadDuration;
      duration[j,process1] = job[j].durations1;
      duration[j,load1] = loadDuration;
      duration[j,unload2] = loadDuration;
      duration[j,process2] = job[j].durations2;
      duration[j,load2] = loadDuration;
      duration[j,unloadS] = loadDuration;
   }
};
```

Fig. 1. The Trolley Problem: Part I.

```
scheduleHorizon = 2000;
UnaryResource machine[Machines];
StateResource trolley(Location);
Activity act[i in Jobs,j in Tasks](duration[i,j]);
Activity makespan(0);

minimize
   makespan.end
subject to {
   forall(j in Jobs & ordered t1, t2 in Tasks)
      act[j,t1] precedes act[j,t2];
   forall(j in Jobs) {
      act[j,process1] requires machine[job[j].machine1];
      act[j,process2] requires machine[job[j].machine2];
   };
   forall(j in Jobs, t in Tasks : t <> process1 & t <> process2)
      act[j,t] requiresState(location[j,t]) trolley;
   forall(j in Jobs)
      act[j,unloadS] precedes makespan;
};

search {
   setTimes(act);
};
```

Fig. 2. The Trolley Problem: Part II.

```
Jobs = {j1,j2,j3,j4,j5,j6};
Tasks = {loadA,unload1,process1,load1,unload2,process2,load2,unloadS};
Location = {m1,m2,m3,areaA,areaS};
Machines = { m1, m2, m3 };
job = [
   <m1,80,m2,60>, <m2,120,m3,80>, <m2,80,m1,60>,
   <m1,160,m3,100>, <m3,180,m2,80>,<m2,140,m3,60> ];
loadDuration = 20;
```

Fig. 3. The Trolley Problem: the Instance Data.

The remaining instructions in Figure 1 specify derived data that are useful in stating the constraints. The instruction

```
Locations location[Jobs,Tasks];
initialize {
    forall(j in Jobs) {
       location[j,loadA] = areaA;
       location[j,unload1] = job[j].machine1;
       location[j,process1] = job[j].machine1;
       location[j,load1] = job[j].machine1;
       location[j,unload2] = job[j].machine2;
       location[j,process2] = job[j].machine2;
       location[j,load2] = job[j].machine2;
       location[j,unloadS] = areaS;
    };
};
```

specifies the locations where each task of the application must take place, while the next two instructions specify the durations of all tasks. The subsequent instructions, shown in Figure 2, are particularly interesting. The instruction

```
ScheduleHorizon = 2000;
```

specifies that the schedule horizon is 2000, i.e., that all tasks must be completed by that date. The instruction

```
UnaryResource machine[Machines];
```

declares the machines of this application. Machines are unary resources, which means that no two tasks can be scheduled at the same time on them. The implementation of `OPL` uses efficient scheduling algorithms for reasoning about these resources, including the edge-finder algorithm [2,6]. Note also that the array `machine` is indexed by a set of values. In fact, arrays in `OPL` can be indexed by arbitrary data structures (e.g., a set of records), which is important to exploit sparsity in large scale applications and to simplify modeling. The instruction

```
StateResource trolley(Location);
```

defines the trolley as a state resource whose states are the five possible locations of the trolley. A state resource is a resource that can only be in one state at any given time: Hence any two tasks requiring a different state cannot overlap in time. The instructions

```
Activity act[i in Jobs,j in Tasks](duration[i,j]);
Activity makespan(0);
```

define the decision variables for this problem. They associate an activity with each task of the application and an activity to model the makespan. An activity in `OPL` consists of three variables, a starting date, an ending date, and a duration, and the constraints linking them. Note also how the subscripts `i` and `j` are used in

the declaration to associate the proper duration with every task. These generic declarations are often useful to simplify problem description. The rest of the statement specifies the objective function and the constraints. The objective function consists of minimizing the end date of the makespan activity. Note that the starting date, the ending date, and the duration of an activity are all accessed as fields of records (or instance variables of objects). The instruction

```
forall(j in Jobs & ordered t1, t2 in Tasks)
    act[j,t1] precedes act[j,t2];
```

specifies the precedence constraints inside a job. It also illustrates the rich aggregate operators in OPL. The instruction

```
forall(j in Jobs) {
    act[j,process1] requires machine[job[j].machine1];
    act[j,process2] requires machine[job[j].machine2];
};
```

specifies the unary resource constraints, i.e., it specifies which task uses which machine. The OPL implementation collects all these constraints that can then be used inside the edge-finder algorithm. The instruction

```
forall(j in Jobs, t in Tasks : t <> process1 & t <> process2)
    act[j,t] requiresState(location[j,t]) trolley;
```

specifies the state resource constraints for the trolley, i.e., it specifies which tasks require the trolley to be at a specified location. The instruction

```
forall(j in Jobs)
    act[j,unloadS] precedes makespan;
```

makes sure that the makespan activity starts only when all the other tasks are completed. Finally, note the instruction

```
search {
    setTimes(act);
};
```

that specifies the search procedure. It illustrates that OPL support user-defined search procedures. The search procedure in this model is rather simple and uses a procedure `setTimes(act)` that assigns a starting date to every task in the array `act` by exploiting dominance relationships. The solution produced by OPL for this application is of the form

```
act[j1,loadA] = [0 -- 20 --> 20]
act[j1,unload1] = [40 -- 20 --> 60]
act[j1,process1] = [60 -- 80 --> 140]
act[j1,load1] = [140 -- 20 --> 160]
act[j1,unload2] = [160 -- 20 --> 180]
```

```
act[j1,process2] = [380 -- 60 --> 440]
act[j1,load2] = [440 -- 20 --> 460]
act[j1,unloadS] = [460 -- 20 --> 480]
...
act[j6,unloadS] = [540 -- 20 --> 560]
makespan = [560 -- 0 --> 560]
```

It displays the starting date, the duration, and the completion time of each activity in the model.

2.2 Transition Times

Assume now that the time to move the trolley from an area to another must be taken account. This new requirement imposes transition times between successive activities. In OPL, transition times can be specified between any two activities requiring the same unary or state resource. Given two activities a and b, the transition time between a and b is the amount of time that must elapse between the end of a and the beginning of b when a precedes b. Transition times are modelled in two steps in OPL. First, a transition type is associated with each activity. Second, a transition matrix is associated with the appropriate state or unary resource. To determine the transition time between two successive activities a and b on a resource r, the transition matrix is indexed by the transition types of a and b.

In the trolley application, since the transition times depend on the trolley location, the key idea is that each activity may be associated with a transition type that represents the location where the activity is taking place. For instance, task `unload1` of job `j1` is associated with state `m1` if the first machine of `j1` is machine 1. The state resource can be associated with a transition matrix that, given two locations, return the time to move from one to the other. The model shown in the previous section can thus be enhanced easily by adding a declaration

```
int transition[Location,Location] = ...;
```

and by modifying the state resource and activity declarations to become

```
StateResource trolley(Location,transition);
UnaryResource machine[Machines];
Activity act[i in Jobs,j in Tasks](duration[i,j])
    transitionType location[i,j];
```

Using a transition matrix of the form

```
[
    [ 0, 50, 60, 50, 90 ],
    [ 50, 0, 60, 90, 50 ],
    [ 60, 60, 0, 80, 80 ],
    [ 50, 90, 80, 0,120 ],
    [ 90, 50, 80,120, 0 ]
];
```

would lead to an optimal solution of the form

```
act[j1,loadA] = [0 -- 20 --> 20]
act[j1,unload1] = [70 -- 20 --> 90]
act[j1,process1] = [90 -- 80 --> 170]
act[j1,load1] = [370 -- 20 --> 390]
act[j1,unload2] = [530 -- 20 --> 550]
act[j1,process2] = [550 -- 60 --> 610]
act[j1,load2] = [850 -- 20 --> 870]
act[j1,unloadS] = [920 -- 20 --> 940]
...
act[j6,unloadS] = [920 -- 20 --> 940]

makespan = [940 -- 0 --> 940]
```

2.3 Capacity Constraints

Consider now adding the requirement that the trolley has a limited capacity, i.e., it can only carry so many items. To add this requirement in OPL, it is necessary to model the trolley by two resources: a state resource as before and a discrete resource that represents its capacity. Several activities can require the same discrete resource at a given time provided that their total demand does not exceed the capacity. In addition, it is necessary to model the tasks of moving from a location to another. As a consequence, each job is enhanced by three activities that represents the move from area A to the first machine, from the first machine to the second machine, and from the second machine to area S. Each of these trolley activities uses one capacity unit of the trolley. The declarations

```
int trolleyMaxCapacity = 3;
DiscreteResource trolleyCapacity(trolleyMaxCapacity);
enum TrolleyTasks {onTrolleyA1,onTrolley12,onTrolley2S};
Activity tact[Jobs,TrolleyTasks];
```

serve that purpose. It is now important to state that these activities require the trolley capacity and when these tasks must be scheduled. The constraint

```
forall(j in Jobs, t in TrolleyTasks)
    tact[j,t] requires trolleyCapacity;
```

specify the resource consumption, while the constraints

```
forall(j in Jobs) {
    tact[j,onTrolleyA1].start = act[j,loadA].start;
    tact[j,onTrolleyA1].end = act[j,unload1].end;
    tact[j,onTrolley12].start = act[j,load1].start;
    tact[j,onTrolley12].end = act[j,unload2].end;
    tact[j,onTrolley2S].start = act[j,load2].start;
    tact[j,onTrolley2S].end = act[j,unloadS].end;
};
```

specify the temporal relationships, e.g., that the activity of moving from area A to the first machine in a job should start when the item is being loaded on the trolley and is completed when the item is unloaded. The trolley application is now completed and the final model is depicted in Figures 4 and 5. This last model in fact is rather difficult to solve optimally despite its reasonable size.

3 The Script Language OPLScript

`OPLScript` is a language for composing and controlling `OPL` models. It is particularly appropriate for applications that require solving several instances of the same model, a sequence of models, or a combination of both as in column-generation applications. See [9] for an overview of these functionalities. `OPLScript` can also be used for controlling `OPL` models in order to find good solutions quickly or to improve efficiency by exploiting more advanced techniques (e.g., shuffling in job-shop scheduling). This section illustrates how `OPLScript` can be used to find a good solution quickly on the final trolley application.

The motivation here is that it is sometimes appropriate to limit the time devoted to the search of an optimal solution by restricting the number of failures, the number of choice points, or the execution time. Figure 7 depicts a script for the trolley problem that limits the number of failures when searching for a better solution. The basic idea of the script is to allow for an initial credit of failures (say, i) and to search for a solution within these limits. When a solution is found with, say, f failures, the search is continued with a limit of $i + f$ failures, i.e., the number of failures needed to reach the last solution is increased by the initial credit. Consider now the script in detail. The instruction

```
Model m("trolley.mod","trolley.dat");
```

defines a `OPLScript` model in terms of its model and data files. Models are first-class objects in `OPLScript`: They can be passed as parameters of procedures and stored in data structures and they also support a variety of methods. For instance, the method `nextSolution` on a model can be used to obtain the successive solutions of a model or, in optimization problems, to produce a sequence of solutions, each of which improves the best value of the objective function found so far. The instruction

```
m.setFailLimit(fails);
```

specifies that the next call to `nextSolution` can perform at most `fails` failures, i.e., after `fails` failures, the execution aborts and `nextSolution()` returns 0. The instructions

```
while m.nextSolution() do {
    solved := 1;
    cout << "solution with makespan: " << m.objectiveValue() << endl;
    m.setFailLimit(m.getNumberOfFails() + fails);
}
```

```
enum Jobs ...;
enum Tasks ...;
enum Location ...;
{Location} Machines = ...;
struct jobRecord {
   Location machine1;
   int durations1;
   Location machine2;
   int durations2;
};
jobRecord job[Jobs] = ...;
int loadDuration = ...;
int transition[Location,Location] = ...;
int trolleyMaxCapacity = ...;

Location location[Jobs,Tasks];
initialize {
   forall(j in Jobs) {
      location[j,loadA] = areaA;
      location[j,unload1] = job[j].machine1;
      location[j,process1] = job[j].machine1;
      location[j,load1] = job[j].machine1;
      location[j,unload2] = job[j].machine2;
      location[j,process2] = job[j].machine2;
      location[j,load2] = job[j].machine2;
      location[j,unloadS] = areaS;
   };
};

int duration[Jobs,Tasks];
initialize {
   forall(j in Jobs) {
      duration[j,loadA] = loadDuration;
      duration[j,unload1] = loadDuration;
      duration[j,process1] = job[j].durations1;
      duration[j,load1] = loadDuration;
      duration[j,unload2] = loadDuration;
      duration[j,process2] = job[j].durations2;
      duration[j,load2] = loadDuration;
      duration[j,unloadS] = loadDuration;
   }
};
```

Fig. 4. The Final Trolley Model: Part I.

```
scheduleHorizon = 2000;
UnaryResource machine[Machines];
StateResource trolley(Location);
DiscreteResource trolleyCapacity(trolleyMaxCapacity);
Activity act[i in Jobs,j in Tasks](duration[i,j])
   transitionType location[i,j];
Activity tact[Jobs,TrolleyTasks];
Activity makespan(0);

minimize
   makespan.end
subject to {
   forall(j in Jobs & ordered t1, t2 in Tasks)
      act[j,t1] precedes act[j,t2];
   forall(j in Jobs) {
      act[j,process1] requires machine[job[j].machine1];
      act[j,process2] requires machine[job[j].machine2];
   };
   forall(j in Jobs, t in Tasks : t <> process1 & t <> process2)
      act[j,t] requiresState(location[j,t]) trolley;
   forall(j in Jobs, t in TrolleyTasks)
      tact[j,t] requires trolleyCapacity;
   forall(j in Jobs) {
      tact[j,onTrolleyA1].start = act[j,loadA].start;
      tact[j,onTrolleyA1].end = act[j,unload1].end;
      tact[j,onTrolley12].start = act[j,load1].start;
      tact[j,onTrolley12].end = act[j,unload2].end;
      tact[j,onTrolley2S].start = act[j,load2].start;
      tact[j,onTrolley2S].end = act[j,unloadS].end;
   };
   forall(j in Jobs)
      act[j,unloadS] precedes makespan;
};

search {
   setTimes(act);
};
```

Fig. 5. The Final Trolley Model: Part II.

```
Model m("trolley.mod","trolley.dat");
int fails := 1000;
m.setFailLimit(fails);
int solved := 0;
while m.nextSolution() do {
   solved := 1;
   cout << "solution with makespan: " << m.objectiveValue() << endl;
   m.setFailLimit(m.getNumberOfFails() + fails);
}
m.setFailLimit(m.getNumberOfFails() + fails);
if solved then {
   m.restore();
   cout << "final solution with makespan: " << m.objectiveValue() << endl;
}

cout << "Time: " << m.getTime() << endl;
cout << "Fails: " << m.getNumberOfFails() << endl;
cout << "Choices: " << m.getNumberOfChoicePoints() << endl;
cout << "Variables: " << m.getNumberOfVariables() << endl;
```

Fig. 6. A Script for the Trolley Problem (`trolley.osc`) .

make up the main loop of the script and produce a sequence of solutions, each of which having a smaller makespan. Note the instruction

```
m.setFailLimit(m.getNumberOfFails() + fails);
```

that retrieves the number of failures needed since the creation of model `m` and sets a new limit by adding `fails` to this number. The next call to `nextSolution` takes into account this new limit when searching for a better solution. Note also the instruction `m.restore()` to restore the last solution found by `nextSolution()`. This script displays a result of the form

```
solution with makespan: 2310
solution with makespan: 2170
solution with makespan: 2080
...
solution with makespan: 1690
solution with makespan: 1650
solution with makespan: 1620
final solution with makespan: 1620
Time: 7.0200
Fails: 3578
Choices: 3615
Variables: 204
```

4 Code Generation in OPL Studio

Once a reasonable model has been successfully designed in OPL, it can be integrated in a larger application through C++ code generation.

```
int main(int argc,char* argv[])
{
   IloSolver_trolleyComplete solver;
   if (solver.nextSolution()) {
      IloArray_act act = solver.get_act();
      IloEnum_Jobs Jobs = solver.get_Jobs();
      IloEnum_Tasks Tasks = solver.get_Tasks();
      IloEnumIterator_Jobs iteJobs(Jobs);
      while (iteJobs.ok()) {
         IloEnumIterator_Tasks iteTasks(Tasks);
         while (iteTasks.ok()) {
            cout << "act[" << *iteJobs << "," << *iteTask << "]=";
            cout << act[*iteJobs][*iteTask] << endl;
            ++iteTasks;
         }
         ++iteJobs;
      }
      cout << endl;
   }
   solver.end();
   return 0;
}
```

Fig. 7. C++ Code for the Trolley Problem.

The basic idea behind code generation consists of producing a C++ class associated with each object in the model and a top-level class for the model. In other words, the generated code is specialized to the model and is strongly typed. These classes can then be used to access and modify the data and, of course, to solve the model and collect the results. Figure 6 depicts C++ code to obtain the first solution to the trolley application and to display the main activities. Instruction `IloSolver_trolley solver;` defines an instance `solver` that encapsulates the functionality of the trolley model. The class definition is available in the `.h` that is not shown here. The instruction `IloArray_act act = solver.get_act();` is used to obtain the array of activities, while the instructions

```
IloEnum_Jobs Jobs = solver.get_Jobs();
IloEnum_Tasks Tasks = solver.get_Tasks();
IloEnumIterator_Jobs iteJobs(Jobs);
```

obtain the enumerated types and define an iterator for the jobs. The remaining instructions iterate over the enumerated sets and display the activities.

Acknowledgments

Pascal Van Hentenryck is supported in part by the *Actions de recherche concertées (ARC/95/00-187)* of the Direction générale de la Recherche Scientifique.

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. J. Carlier and E. Pinson. Adjustments of Heads and Tails for the Job-Shop Problem. *European Journal Of Operations Research*, 78:146–161, 1994.
3. A. Colmerauer. An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418, 1990.
4. 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.
5. R. Fourer, D. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.
6. W. Nuijten. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. PhD thesis, Eindhoven University of Technology, 1994.
7. Ilog SA. Ilog Solver 4.4 Reference Manual, 1998.
8. G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*. LNCS, No. 1000, Springer Verlag, 1995.
9. P. Van Hentenryck. OPL Script: A Language for Composing and Controlling OPL Models. Technical Report RR 3/99, Department of Computing Science and Engineering, UCL, April 1999.
10. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.
11. P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint Programming in OPL. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Paris, France, September 1999.