OPL Script: Composing and Controlling Models

Pascal Van Hentenryck[1] and L. Michel[2]

[1] Université catholique de Louvain
2 Place Sainte-Barbe, 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@ilog.fr

Abstract. This paper is a brief introduction to OPLScript, a script language for composing and controlling optimization models. OPLScript makes it possible to state concisely many applications that require solving several instances of the same model (e.g., to perform some sensivity analysis), a sequence of models, or a combination of both as in column-generation applications. OPLScript also enables modellers to exercise some control over a model and/or to use traditional scripting facilities. The basic abstractions of OPLScript are the concepts of models and abstract models that make it possible to develop, maintain, test, and reuse models independently of the scripts using them and to develop scripts that apply to a variety of models.

1 Introduction

The last decades have witnessed the development of many modeling languages for mathematical programming (e.g., [4,2]). These languages significantly decrease the development time of optimization applications by providing high-level algebraic and sets notations and data modeling facilities. Recent modeling languages such as OPL [8] further enhance these functionalities by supporting constraint programming notations and technologies and by letting modellers specify their search procedures for combinatorial optimization problems.

Many practical applications however involve more than just solving a model, even when their combinatorial aspect is considered in isolation. Typically, these applications require solving several instances of a model (e.g., to perform some sensitivity analysis), a sequence of models where the solutions of one model are the inputs of the next model, or a combination of both as in column-generation applications. In addition, some of these applications may need to exercise some control over the search procedure to improve performance, to obtain a reasonable solutions quickly, or, more generally, to tailor a general model to a given application.

This paper is a brief introduction to OPLScript, a script language for composing and controlling optimization models. By combining high-level data modeling

K.R. Apt et al. (Eds.): New Trends in Constraints, LNAI 1865, pp. 75–90, 2000.

facilities of modeling languages with novel abstractions for composing and controlling models (e.g., abstract models, bases, ...), OPLScript simplifies significantly the implementation of these applications. In addition, by encapsulating the generic aspects of models, OPLScript makes it possible to use traditional scripting functionalities over models (e.g. iterating over a collection of models or implementing generic search procedures). Finally, by abstracting irrelevant aspects of models, OPLScript makes it easy to combine models using different implementation technologies (e.g., mathematical and constraint programming).

To stress the contributions of OPLScript, it is interesting to contrast it with several tools, including AMPL, constraint programming languages, and traditional script languages.

- The procedural extensions of AMPL were motivated by similar considerations but OPLScript provides both novel functionalities and higher-level abstractions. These features endow OPLScript with traditional scripting functionalities and the ability to define generic search strategies, both of which are outside the scope of AMPL. In addition, the resulting separation of concerns and encapsulation make it possible to develop, test, and maintain models independently from the scripts using them. Finally, OPLScript supports the composition of models implemented by fundamentally different technologies (e.g., mathematical and constraint programming).
- The applications described in this paper can already be implemented in constraint programming languages such as CHIP [3], CLAIRE [5], ECLIPSE [9], the Ilog optimization suite [6], and OZ [7]. This is of course not surprising since these languages are Turing-complete. The main contribution of OPLScript is to simplify further the development of these applications by combining traditional data modeling facilities of modeling languages with novel abstractions for composing and controlling models. These abstractions relieve modellers from tedious design and implementation effort and let them focus on high-level modeling issues.
- OPLScript also goes far beyond traditional scripting languages by providing very-high level data modeling and control features and by providing abstractions that are specific for optimization applications. In this context, OPLScript is best viewed as a domain-specific script language for optimization.

The rest of this paper highlights some of the functionalities of OPLScript by presenting some small, but representative, applications. Section 2 is a short presentation of the main concepts by means of simple examples. It introduces models and to use them to find solutions and/or optimal solutions and open arrays to store an unknown number of solutions or configurations in column-generation applications. The next three sections describe a script to solve several instances of a production model, a script to solve the so-called Vellino problem that sequences two models using different technologies, and a script for a column-generation application. These sections introduce functionalities such as the ability to import data between scripts and models and linear programming bases to achieve more performance in column-generation applications. Section 6 describes more

traditional script fonctionalities: it introduces the concept of abstract models and the ability to control models in OPLScript. Section 7 concludes the paper.

2 Getting Started

Since OPLScript is a script language for composing and controlling models for optimization problems, it is thus not surprising that models are a fundamental concept of the language. Consider the script

```
Model m("nqueens.mod","nqueens.dat");
if m.solve() then
    forall(i in 1..m.n) {
       cout << "queens[" << i << "] = ";
       cout << m.queens[i] << endl;
    }
```

that displays a solution to a queens problem. It uses the OPL model file `nqueens.mod` (see Figure 1) and the data file `nqueens.dat` to declare a model `m`. The instruction `m.solve()` solves the model, i.e., in this case, it finds a solution to the model. The rest of the statement uses the model objects to display the solution. Since models in OPLScript have their own scope, the various data declared in `nqueens.mod` are accessed through `m` in the same way as fields of records. In particular, `m.queens[3]` denotes the value of `queens[3]` in `nqueens.mod`.

```
int n = ...;
range Domain 1..n;
var Domain queens[Domain];
solve
   forall(ordered i,j in Domain) {
      queens[i] <> queens[j];
      queens[i] + i <> queens[j] + j;
      queens[i] - i <> queens[j] - j
   };
```

Fig. 1. The n-Queens Model (`nqueens.mod`) .

Consider now the script

```
Model m("nqueens.mod","nqueens.dat");
int n := 0;
while m.nextSolution() do
    n := n + 1;
cout << "Number of Solutions: " << n << endl;
```

that illustrates how to compute and display the number of solutions to the queens problem. It uses the instruction `m.nextSolution()` that computes successive solutions to the model `m`. More precisely, the first call to `nextSolution` computes the first solution, the second call the second solution, and so on until all solutions have been found. For optimization problems, instruction `nextSolution` can be used to obtain successive solutions, each of which improving the best value of the objective function found so far. More precisely, the first call computes the first solution, the second call computes another solution with a better objective value and so on until no solution improving the best solution obtained so far is found. For instance, the script

```
Model m("bridge.mod","bridge.dat");
while m.nextSolution() do
    cout << "Objective value: " << m.objectiveValue() << endl;
```

displays the objective value of the successive solutions found to the bridge problem [1]. To obtain the optimal solution, it is sufficient to enhance the script by using the `restore` method on the model:

```
Model m("bridge.mod","bridge.dat");
while m.nextSolution() do
    cout << "Objective value: " << m.objectiveValue() << endl;
m.restore();
cout << "Objective value: " << m.objectiveValue() << endl;
```

This method restores the last solution found by `nextSolution`, thus retrieving the last solution found so far. For linear programs, of course, method `nextSolution` returns the optimal solution directly. In optimization problems, the optimal solution of a model can be obtained directly by using method `solve()` as in

```
Model m("bridge.mod","bridge.dat");
if m.solve() then
    cout << "Objective value: " << m.objectiveValue() << endl;
```

Assume now that we are interested in storing all the solutions of the queens problem. A possible, but inefficient, solution consists of counting the solutions, declaring an array of the appropriate size, and then filling the array by solving the problem again. A more elegant and efficient solution consists of using an open array. An open array is an array that grows or shrinks dynamically at runtime. The script depicted in Figure 2 illustrates how open arrays help solve the problem. The instruction

```
Open int solution[1..0,1..m.n];
```

defines a 2-dimensional open array of integers, i.e., an open array of open arrays of integers. The first dimension is an empty range in the declaration and hence the open array is empty as well. When a solution is found, the method call `solution.addh()` increases the upper bound of the open array `solution` by one (`addh` stands for "add high"). The first time this instruction is executed in the above script, the array `solution` becomes an array of one element whose only index is 1 and whose value is an array that has not yet been initialized.

```
Model m("queens.mod","queens.dat");
int nb := 0;
Open int solution[1..0,1..m.n];
while m.nextSolution() do {
   n := n + 1;
   solution.addh();
   forall(i in 1..m.n)
      solution[n,i] := m.queens[i];
}
forall(s in 1..n) {
   forall(i in 1..m.n)
      cout << "queens[" << i << "] = " << solution[s,i] << endl;
   cout << endl;
}
```

Fig. 2. A Script to Store All Solutions (`storequeens.osc`) .

The instructions

```
forall(i in 1..m.n)
    solution[n,i] := m.queens[i];
```

initializes this dynamically created array in the traditional way. The remaining instructions display all solutions.

It is important to stress that a model is best viewed as an object and its data, variables, and constraints can be tought of as instance variables. After a model has been solved, its results can be accessed as in OPL. For instance, it is possible to obtain their values, the slack of the constraints, the dual variables in the case of linear programs, and any other results provided by OPL.

3 Solving Several Instances of a Model

Practical applications often require to solve several instances of a model in order to perform various forms of sensibility analyses or to execute "what-if" scenarios on a model. This section illustrates how such an application can be expressed in OPLScript using a simple production planning model. The model is not shown for brevity, since it is not necessary to understand its detail to follow the discussion. It suffices to know that products can be produced either inside or outside the factory. The inside production is cheaper but is limited by the resources (capacity constraints). Consider the script

```
Model produce("mulprod.mod") editMode;
import enum Resources produce.Resources;
int+ capFlour := produce.capacity[flour];
```

```
forall(i in 1..4) {
    produce.capacity[flour] := capFlour;
    produce.solve();
    cout << "Objective Function: " << produce.objectiveValue() << endl;
    produce.reset();
    capFlour := capFlour + 1;
}
```

The instruction

```
Model produce("mulprod.mod") editMode;
```

declares the model in edit mode, which makes it possible to edit the instance data. The next instruction imports the enumerated type `Resources` from the model to the script in order to simplify the rest of the statement (i.e., it enables the script to use `flour` instead of `produce.flour`. The loop of the script solves 4 instances of the model that differ by the capacity of resource `flour` which is incremented by one at each iteration.

This model is a linear program and hence it may be appropriate to use the optimal basis of an instance as a starting point for solving another instance. Such a basis represents a vertex of the polyhedron defined by the linear program and starting from a good vertex may speed up the linear program significantly. OPLScript supports this functionality by providing the concept of linear programming basis. The loop of the script depicted previously can be replaced by the instructions

```
forall(i in 1..4) {
    produce.capacity[flour] := capFlour;
    produce.solve();
    cout << "Objective Function: " << produce.objectiveValue() << endl;
    Basis b(produce);
    produce.reset();
    produce.setBasis(b);
    capFlour := capFlour + 1;
}
```

to take advantage of this facility. The instruction `Basis b(produce)` declares a basis and initializes it with the optimal basis of model `produce`. The instruction `produce.setBasis(b)` specifies that basis `b` must be used as a starting basis when solving model `produce`. This use of bases may significantly decrease the number of iterations for linear programs. As shown later in this paper, OPLScript also support partial bases, i.e., bases that only specify the basis status of some of the variables and constraints, which is particularly useful when solving a sequence of problems where the number of variables and/or constraints increase dynamically.

It is important to step back at this point and stress some important contributions of OPLScript here. First, note that (partial) linear programming bases are first-class objects that represent fundamental abstractions for mathematical programming. By providing such abstractions, OPLScript makes it possible to support efficiently fundamental techniques from mathematical programming. Second, note that such extension do not change the nature of the models. A model

can still be viewed as an object that simply has an additional instance variable to store a linear programming basis. The basis, if it exists, is used as a starting point to solve the model.

Finally, it is interesting to show a final version of the loop where the dual values are used to control the number of iterations. Consider the instructions

```
repeat {
    produce.capacity[flour] := capFlour;
    produce.solve();
    cout << "Objective Function: " << produce.objectiveValue() << endl;
    float sumDual := sum(t in produce.Periods) produce.cap[flour,t].dual;
    if sumDual = 0 then
       break;
    Basis b(produce);
    produce.reset();
    produce.setBasis(b);
    capFlour := capFlour + 1;
} until 0;
```

The instruction

```
float sumDual := sum(t in produce.Periods) produce.cap[flour,t].dual;
```

computes the summation of the dual values associated with the capacity constraints. In the model, `cap` is a 2-dimensional array of capacity constraints indexed by the resources and by the time periods. When this summation is zero, the loop terminates since it means that the objective function cannot be improved.

4 Sequences of Models

The previous section has illustrated how to solve several instances of the same problem. This section shows how to solve an application consisting of a sequence of two models: a constraint programming model and an integer program. The application is a configuration problem, known as Vellino's problem, that is a small but good representive of many similar applications. For instance, complex sport scheduling applications can be solved in a similar fashion.

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

the components: wood requires plastic, glass excludes copper and copper excludes plastic. In addition, we are given an initial capacity for each bin, i.e., red bins have a capacity of 3 components, blue bins of 1 and green bins of 4 and a demand for each component, i.e., 1 glass, 2 plastic, 1 steel, 3 wood, and 2 copper components.

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

Fig. 3. A Script to Solve Vellino's Problem (`vellino.osc`) .

The strategy to solve this problem consists of generating all the possible bin configurations and then to choose the smallest number of them that meet the demand. This strategy is implemented using a script `vellino.osc` depicted in Figure 3 and two models `genBin.mod` and `chooseBin.mod` depicted in Figures 4 and 5. It is interesting to study the script in detail at this point. The instructions

```
Model bin("genBin.mod","genBin.dat");
import enum Colors bin.Colors;
import enum Components bin.Components;
```

declare the first model and import the enumerated types; 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.

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

Fig. 4. Generating the Bins in Vellino's Problem (`genBin.mod`) .

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

Fig. 5. Choosing the Bins in Vellino's Problem (`chooseBin.mod`) .

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`. Once this step is completed, the second model is executed and produces a solution at cost 8.

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

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

It is important to stress to both models can be developed and tested independently since import declarations can be initialized in a data file when a model is run in isolation (i.e., not from a script). This makes the overall design compositional. Note also that this model cannot be stated in `AMPL` which does not support sequences of solutions.

5 Column Generation

The last two sections have shown how to solve several instances of the same model and a sequence of distinct models. This section illustrates how to express a column-generation algorithm, Gilmore and Gomory's cutting stock algorithm, that combines and expands these functionalities. In particular, the script uses partial linear programming bases to use the optimal basis of an instance as the starting point for a (slightly) larger instance. This feature, combined with the compositionality and encapsulation provided by OPLScript, makes this script more elegant than the traditional AMPL script.

The problem consists of cutting big wood boards into small shelves to meet the customer demands while minimizing the number of boards used. A given instance specifies the length of the boards (an integer), the length of the shelves (integers), and the demand for each shelf type (integers as well). The complexity in this application is that it is not practical to generate all the possible configurations: in typical applications, there are so many possible cutting configurations that they cannot even be stored because the boards are large and there are many shelves whose sizes may be rather small. The basic idea behind the column generation approach consists of starting with a set of configurations

```
data "cut.dat";

int+ boardLength := ...;
int nbShelves := ...;
range Shelves 1..nbShelves;
int shelf[Shelves] := ...;
int nbConfig := -1;
Open int+ config[0..nbConfig,Shelves];
forall(i in Shelves) {
   nbConfig := nbConfig+1;
   config.addh();
   forall(j in Shelves)
      config[nbConfig,j] := 0;
   config[nbConfig,i] := boardLength/shelf[i];
}
Model chooseC("chooseConfigs.mod","cut.dat");
Model generateC("generateConfigs.mod","cut.dat");
repeat {
   chooseC.solve();
   Basis b(chooseC);
   forall(i in Shelves)
      generateC.cost[i] := chooseC.meet[i].dual;
   generateC.solve();
   if generateC.objectiveValue() > 1 then {
      nbConfig := nbConfig + 1;
      config.addh();
      forall(j in Shelves)
         config[nbConfig,j] := generateC.use[j];
      chooseC.reset();
      chooseC.setBasis(bc);
      generateC.reset();
   } else
      break;
} until 0;
range Configs 0..nbConfig;
int cuti[p in Configs] := ftoi(ceil(chooseC.cut[p]));
int obj := sum(p in Configs) cuti[p];
cout << "Solution with Cost: " << obj << endl << endl;
```

Fig. 6. A Script For Gilmore and Gomory's Cutting Stock (`gomory.osc`).

and then to use the optimal solution to the (linear relaxation of the) simplified problem to generate a new configuration that is promising. In this cutting stock application, a column generation consists of solving a knapsack problem: the capacity constraint makes sure that a legal configuration is generated, while the objective function aims at finding a configuration whose associated variable would enter the basis. This process is iterated until no column can be generated. The upward rounding of the solution is generally considered of sufficient quality.

```
int+ boardLength = ...;
int nbShelves = ...;
range Shelves 1..nbShelves;
int shelf[Shelves] = ...;
int demand[Shelves] = ...;
import int nbConfig;
import int+ config[0..nbConfig,1..nbShelves];
range Configs 0..nbConfig; constraint meet[Shelves];
var float+ cut[Configs];
minimize
   sum(j in Configs) cut[j]
subject to
   forall(i in Shelves)
      meet[i]: sum(j in Configs) config[j,i] * cut[j] >= demand[i];
```

Fig. 7. A Model for Choosing Cutting Stock Configurations (`chooseConfigs.mod`) .

Figure 6 depicts a script for this application, while Figures 7 and 8 show the models. There are a number of interesting features of OPLScript illustrated in this application, as will become clear shortly. The instruction `data "cut.dat"` specifies that the data initializations needed by the script may be found in the file `cut.dat`. The first set of instructions declare an open array `config` to store the configuration and initializes it with a first set of configurations, one for each shelf. The second loop is the core of the script. The first two lines

```
chooseC.solve();
Basis b(chooseC);
```

select the optimal set of configurations from the subset available and store the basis for future reuse. The lines

```
forall(i in Shelves)
    generateC.cost[i] := chooseC.meet[i].dual;
generateC.solve();
```

use the dual values to initialize the objective function of the knapsack model, since these values correspond to the simplex multipliers. The knapsack model is

```
int+ boardLength = ...;
int nbShelves = ...;
range Shelves 1..nbShelves;
int shelf[Shelves] = ...;
int demand[Shelves] = ...;
Open float cost[Shelves];
var int use[Shelves] in 0..boardLength;
maximize
   sum(i in Shelves) cost[i]*use[i]
subject to
   sum(i in Shelves) shelf[i] * use[i] <= boardLength;
```

Fig. 8. A Model for Generating New Cutting Stock Configurations (`generateConfigs.mod`)

then solved. If its objective value is greater than one, then its reduced cost is negative and thus the variable associated with the generated column will enter the basis. The lines

```
nbConfig := nbConfig + 1;
config.addh();
forall(j in Shelves)
    config[nbConfig,j] := generateC.use[j];
chooseC.reset();
chooseC.setBasis(bc);
generateC.reset();
```

add a new configuration using the result of the knapsack, reset the models, and use the basis saved previously as starting point for finding a new solution to model `chooseC` that now has an additional variable. The implementation takes care of extending this partial basis into a full basis for the extended problem. The script terminates when the objective value of the knapsack problem is not greater than 1, in which case the linear programming solution is provably optimal. The final solution is obtained by rounding up this solution. The two models should not raise much difficulty at this point. Note that they can be developed and tested in isolation once again.

6 Controlling Models

In practical applications, it is often desirable, not only to specify search procedures as allowed in OPL, but also to control them in order to, say, obtain solutions quickly or to produce "good" solutions within reasonable time. This section discusses briefly some of the OPLScript support in this area. It is useful to specify that

the control discussed in this section is orthogonal to the control of search procedures. Search procedures and search strategies (e.g., depth-first search, limited-discrepancy search, best-first search) can be expressed directly in the OPL model and the control discussed here is a form of meta-strategy.

```
Model m("trolley3.mod");
int fails := 1000;
m.setFailLimit(fails);
int solved := 0;
while m.nextSolution() do {
   solved := 1;
   m.setFailLimit(m.getNumberOfFails() + fails);
}
if solved then {
   m.restore();
   cout << "final solution with makespan: " << m.objectiveValue() << endl;
}
```

Fig. 9. A Script for the Trolley Problem (`trolley.osc`) .

Consider the idea of limiting the time devoted to the search of an optimal solution by restricting the number of failures, the number of choice points, the execution time, or the number of discrepancies between a heuristic and the search procedure. Figure 9 depicts a script for an advanced scheduling problem that limits the number of failures when searching for a better solution. Similar scripts can be written to control execution time or to control a limited discrepancy search. 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. 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 instruction

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

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.

Figure 10 shows how to encapsulate this strategy, since it is essentially model-independent and can be applied to other models as well. The script features a

```
procedure int limitedFailureSearch(AbstractModel m, int fails)
{
   m.setFailLimit(fails);
   int solved := 0;
   while m.nextSolution() do {
      solved := 1;
      m.setFailLimit(m.getNumberOfFails() + fails);
   }
   if solved then
      m.restore();
   return solved;
}

Model trolley("trolley3.mod");
int solved := limitedFailureSearch(trolley,1000);
if solved then
   cout << "final solution with makespan: " << m.objectiveValue() << endl;
```

Fig. 10. The Script for the Trolley Problem Revisited (`abstract.osc`) .

procedure declaration whose first parameter is an abstract model. In the script, the first parameter receives the trolley model but it could be used for other models as well.

The class of abstract models is a superclass of the model class that encapsulates all the generic aspects of models. Abstract models are first-class objects that can be passed to procedures, be assigned to other abstract models or models, and stored in data structures such as sets, arrays, and records. Abstract models give a dimension to OPLScript that is missing from AMPL. They make it possible to solve the same model for various data files, to run several models with the same data files, or, more simply, to execute benchmarking sequences. For instance, it is possible to write a script of the form

```
setof(string) Models := ...;
forall(m in Models) {
    AbstractModel a := Model(m,"inst.dat");
    a.solve();
    cout << "Time: " << a.getTime() << endl;
}
```

that compares the behaviour of several models on the same instance data.

7 Conclusion

This paper was a brief introduction to OPLScript, a script language for composing and controlling optimization models. OPLScript was primarily motivated by our

desire to simplify the solving of applications that requires solving several instances of the same model, sequences of models, or a combination of both as in column-generation algorithms. By combining high-level data modeling facilities of modeling languages with novel abstractions (e.g., models, abstract models, basis) to compose and control models, OPLScript may reduce the development time of these applications substantially by letting modellers focus on high-level modeling issues and relieving them from tedious design and implementation effort. OPLScript also provides a clear separation of concerns between scripts and models, allowing models to be developed, maintained, and tested independently from the scripts using them. Finally, OPLScript provides some traditional scripting functionalities over models, e.g., the ability to iterate over a collection of models and/or to define procedures whose parameters are abstract models. Future work on OPLScript will focus on enhancing the traditional scripting functionalities and on code generation.

Acknowledgments. Comments from the reviewers help improve the presentation. Thanks also to Irvin Lustig, Janet Lowe, and Jean-François Puget for interesting discussions on this paper.

References

1. M. Bartusch. *Optimierung von Netzplaenen mit Anordnungsbeziehungen bei Knappen Betriebsmitteln.* PhD thesis, Fakultaet fuer Mathematik und Informatik, Universitaet Passau, Germany, 1983.
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. 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. 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.
6. Ilog SA. Ilog Solver 4.31 Reference Manual, 1998.
7. G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today.* LNCS, No. 1000, Springer Verlag, 1995.
8. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.
9. M. Wallace, St. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.