A Short Introduction to OPL - Optimization Programming Language

Roland Martin
Technische Universität Darmstadt
Fachbereich für Informatik
Fachgruppe Algorithmik

September 2002

Chapter 1

What is OPL ?

1.1 Overview

In a nutshell OPL is a modeling language for mathematical programming and combinatorial optimization problems. OPL was the first language that integrates an expressive constraint language and the ability to specify search procedures and strategies in high-level algebraic and set notations from mathematical modeling languages.
The essence of constraint programming is a two-level architecture integrating a constraint component and a programming component.
The constraint component on the one hand provides the basic operations of the architecture. It consists of a system that reasons about fundamental properties of constraint systems such as satisfiability and entailment of constraints.
The programming component on the other hand, provides algorithms and strategies to find a solution to the problem.
The programming-language component is the interface to the constraint component. It parses the OPL input and sets up the programming component accordingly.
Moreover, OPL lets users specify search procedures tailored to the problem at hand. Hereby it improves the expressivness of traditional modeling languages by offering concepts such as higher-order constraints, logical combination of constraints, global constraints, etc.
OPL also supports extensions of the language which provide scheduling-specific features like activities and resources. These features are commonly-used in real-world problems and are mainly used by the ILOG Scheduler.
The following introduction to OPL is motivated by the book of van Hentenryck [1].

1.2 Syntax of OPL

In this section we will see the main concepts of the language OPL. We will learn about the structure of OPL models and the specific language constructs like the data types and the output.
As a brief preview of the simplicity of OPL consider the following example.
The typical LP:
Maximize $c^t x$

subject to $\sum_{j=1}^{n} d_{ij} \leq s \quad (1 \leq i \leq n)$

would be written in OPL:

```
maximize
  sum(i in 1..n) c[i]*x[i]
subject to
  forall (i in 1..n) sum(j in 1..n) d[i,j] <= s;
```

The keyword `maximize` with the next line in this short example is the objective function. If the optimization is achieved by minimizing the objective value then the keyword `minimize` is used instead.
The keywords `subject to` indicates the begin of the constraint store. In the constraint store all constraints are declared. If there is more than one constraint, brackets indicate where the constraint store starts (exactly after the keywords `subject to`) and where it ends (after the last constraint).
The keyword `forall` saves us a lot of code. For all elements `i` in the set `1..n` (in this small example) the following instructions are executed. Without `forall` the constraint store would have n lines of constraints,
from `sum(j in 1..n) d[1,j] <= s;`
to `sum(j in 1..n) d[n,j] <= s;`.

1.2.1 Structure of an OPL model

An OPL model consists of the following sections:

- declaration of the constants and variables (Section 1.2.2)
- the optional objective function (Section 1.2.2)
- the constraint store (Section 1.3)
- customization of the search procedure (Section 1.4)

Declaration of the input data and variables

This section is concerned with the variables for which values out of their domains are searched as well as with the initialization of all data needed for the model. The user specifies if she wants to group the initialization data in a special file or initialize the data directly in the model.

The objective function

In this section the user specifies the function to determine the quality of a solution. The user specifies if she wants to minimize or maximize the value of the objective function. If the problem is a pure constraint satisfaction problem, that is no optimization is needed, the objective function may be omitted. In this case the keyword `solve` is used instead of `maximize ... subject to`.

The constraint store

In this section the user specifies all constraints that must be fulfilled for a solution to be feasible.

Customization of the search procedure

In this section the user specifies the customizations of the default search strategies if any is wanted. The default variable and value ordering strategy is very simple. Usually the variables are instantiated in the order in which they are in the declaration part of the model and the values are assigned beginning with the first in the domain. But the user may wish to have some variables instantiated before others because she has specific knowledge of the solution.
In this section of the model the user can alter the variable and value ordering or make specific assignments to variables.

> **Example:** Consider in an automated manufacturing model the objective is to find an assignment of mounting jobs to machines. Since there are components that are very expensive one may want to mount them on the last machine. That reduces the danger of the component being damaged during the mounting process. Then the user can assign this job to the last machine in the line by stating this in this section of the model.
>
> Moreover the user knows that another specific job is more likely to be mounted on the first half of the machines (in an optimal solution). Then she could specify the search such that this specific job is assigned to a machine in the first half. Only in case there is no (optimal) solution with such an assignment other machines will be assigned to this job.

1.2.2 Data Types and Initialization

OPL supports the basic data types

- integer `int`
- floating point numbers `float`
- enumerated types `enum`

Integers and floats have the same features as in many programming languages and need no further explanation. It is also possible to declare nonnegative integers and floats. This is done by adding a `+` to the expression (`int+, float+`). This declaration is used for variables (introduced in Section 1.2.3) and will be explained further in that section. Enumerated types are similar to the analogous concepts in C or PASCAL and are specified by listing the values of their domain.
The benefit of an enumerated type is that it produces more readable programs.

Code examples:

```
int i =3;
float pi = 3.14;
enum Days {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

It is also possible to perform operations on basic data types. Floats and integers can be added, substracted etc. (If a float and an integer are to be added, the value will be of the type `float`. To convert a float into an integer, the operator `ftoi` is needed.
(Code example: `int smallpi ftoi(pi);`).

The most common operators are listed below. For a complete list reference the book of van Hentenryck [1]

- integer operators
  - `mod`: modulo computation
  - `abs`: absolute value
  - `maxint`: the largest constant expressable by OPL
- float operators
  - `sqrt`: the square–root of its argument
  - `ceil`: returns the smallest integer greater than or equal to its argument
  - `floor`: returns the largest integer smaller than or equal to its argument
  - `distToInt`: the distance to the nearest integer from its argument
  - `frac`: returns the fractional part of its argument
  - `trunc`: returns the integer part of its argument
  - `infinity`: a constant that represents ∞
- enum operators
  - `first`: returns the first enumerated value of its argument
  - `last`: returns the last enumerated value of its argument
  - `card`: returns the cardinality of its argument
  - `ord`: returns the position of the argument in its enumerated type (the argument is a value of an enumerated type)
  - `next`: returns the successor of the argument
  - `prev`: returns the predecessor of the argument

Code examples:

```
int j = i + 6;
float size = j * pi;
int smallpi ftoi(floor(pi));   // Result: smallpi = 3
float root = sqrt(j);   // Result: root = 3.0
float dimension = infinity;
```

```
enum Days {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
int interval = card(Days);   // Result: interval = 7
Days f = first(Days);   // Result: f = Monday
Days middle = next(Tuesday);   // Result: middle = Wednesday
int position = ord(Saturday);   // Result: position = 6
```

More complex data types can be built from the basic data types in OPL. The provided building concepts are: ranges, arrays, records, and sets.

Ranges

Ranges indicate the domain of variables and are useful for writing readable programs and restrict variables.
A range is specified by its lower and upper bound where these bounds can be given in expressions. In this case the bounds have to be in brackets as seen below.

Code examples:

```
range float Length 1.4..2.3;
range int Rows 1..10;
int i =5;
range Colums [i+3..2*i];
```

Arrays

OPL supports multidimensional arrays and arrays with implicite initialization. Implicite initialization means that the arrays are not initialized by concrete values but by mathematical expressions. (Code example: `int value[i in 0..10,j in 1..3] = 10*i+j;`).
(Note: In the book of van Hentenryck [1] the author calls implicite initialized arrays generic arrays. Since the expression generic is nowadays heavily overloaded we will not use it and call these arrays implicite initialized, like we did it above).
This is very useful when the data of an array obeys the laws of a function. Consider an array with values that are twice their position index. Then the array would be initialized with the mathematical expression `2*i` where `i` is the index position in the array. Arrays can be of a basic type or of a different data type. Therefore, it is possible to build an array of arrays (or further called multidimensional array) or an array of structs etc.

Code examples:

```
int vector[1..3] =...;
float b[1..2] = [1.3 , 5.8];
Days d[3,4] =[Wednesday, Thursday];

range length 1..1000;
int even[i in length] = 2*i;
```

```
range dim1 1..3;
range dim2 1..2;
int vector1[dim1] =[2,4,6];
int array1[dim1,dim2] =...;
int array2[j in dim2,i in dim1] = m[i,j];
int value[i in 0..10,j in 1..3] = 10*i+j;
```

The example of array2 is a permutation of the indices of array1. This demonstrates as an example that implicite initianlized arrays can be used to transpose matrices very easily. Implicite initialized arrays should be used whenever possible, since they make the model more explicit and readable.
Consider the situation like the array `even`. It would be very tedious to write all values down.

Records (`structs`)

Records cluster related data together and they are very useful for modeling logical units that belong together. That way objects can be modeled easily where the data in the records correspond to the attributes of the object.

Code examples:

```
struct Car {
   int ccm;
   int kW;
   int color;
   int type;
   int price;
};

struct Point {
   int x;
   int y;
};

struct Rectangle {
   int id;
   Point p[2];
};
```

To instantiate a record the user first has to declare its type (like seen in the examples above). Then she declares an input variable of this type and instantiates it.
To get access to a field of a record one has to name this field by the record's name and the field name separated by a dot (`recordname.fieldname`).

Code examples:

```
Point Left_Lower =<2,3>;
Point Right_Upper =<7,5>;

Rectangle Room =<3,[<2,3>,<7,5>]>;

int coord = Left_Lower.x;
int room_id = Room.id;
```

Sets

OPL also supports sets of arbitrary types to model data. If `T` is a type, then {`T`} is a set of the type `T`.

Code examples:

```
struct Car {
   int ccm
   int kW
   int color
   int type
   int price
};

{Car} Cars_In_Stock =...;

{Point} Points_In_Rectangle ={<1,2>,<2,2>,<2,1>,<0,0>,<-1,2>};
```

In sets, data which is equal, e.g. numbers, are overwritten since they are already in the set. This means adding the value 2 to a set which already has this value does not change the set.
Like on the basic data types the user can perform operations on the more complex data types introduced in this section.
The most common operations are listed below. Again, for a complete list reference the book of van Hentenryck [1].

- aggregate operators
  - `sum`: returns the sum of its arguments
  - `prod`: returns the product of its arguments
  - `min/max`: returns the minimum/maximum of its arguments
- set operators
  - `union`: returns the union of two sets
  - `inter`: returns the intersection of two sets

- relations in expressions

  Since OPL views every relation as a 0-1 integer it is possible to build expressions in terms of relations.

  This means that, in case of aggregation, only values are aggregated, for which some relations hold (e.g. the values must be even).

  Boolean connectives can be used to combine relations. Two of these connectives are `not` (negation) and `=>` (implification). When the left side of the implification is evaluated to 1, then the right side is also evaluated to 1.

Code examples:

```
range length 1..1000;
int all = sum(i in length) even(i);

range dim1 1..3;
range dim2 1..2;
int biggest = max(i in dim1, j in dim2) array2[i,j];

{int} set1 = {1,2,3,4};
{int} set2 = {3,4,5,6};
{int} u = set1 union set2;   \\Result: u = {1,2,3,4,5,6}
{int} i = set1 inter set2;   \\Result: i = {3,4};

{int} s[0..10] = ...;
{int} occur[i in 0..10] = sum(j in 0..n) (s[j] = i);
        // Result: occur[i] is initialized with the number of
        // times i occurs in the array s

range size -100..100;
size numbers[1..50] =...;
int amount = sum(i in 1..50) (numbers[i] < 80 & numbers[i] > -40);
        // Result: amount summarizes all occurrences of numbers
        // between -40 and 80 in the array numbers.

forall (i in 1..10)
  m[i] <> 0 => m[m[i]] = i;
```

Relations in Expressions

OPL views every relation as a 0-1 integer. Therefore expressions can be constructed in terms of relations.

Data initialization

OPL supports different forms of initialization. Two of the concepts are inline and file initialization. Inline initialization takes place during declaration of the variables:

`int a[1..3] = [3,6,8];`
In file initialization the data is given in a specified file:
`int a[1..3] < "data1.dat";`.
Initializing the data in a separate file has the advantage that it can be different for various problem instances and makes the file more readable.
In the case of inline initialization there is a command `initialize`. One have to use it, in case the values for the constants have to be computed and cannot all at once given in one statement.
Consider an array like `array1[0..5,0..5]` that has to be initialized with different values but to much to be written down exhaustively. On the other hand there is no function to initilize all entries. With the `initialize` statement, the entries can be initialized successively.
Code example:

```
int array1[0..5,0..5];

initialize
{
  forall (i in 0..5)
  {
    array2[i,0] = 0;
    array2[0,i] = array2[i,0];
  };

  forall(i,j in 1..5)
    array2[i,j] = i*j;
};
```

1.2.3 Output Variables

In OPL we differentiate between the input data (constants) and the output data (variables). The input data is the specification of the model and the output data specify the solution. Variables are like constants but are specified by the keyword `var` during their declaration.
It is the purpose of an OPL model to find values for the variables such that all constraints are satisfied.
Variables in OPL can be of different types such as arrays and even multidimensional arrays.

Code examples:

```
var int Price in 1..100;
var float x_Coord in -10..10;
var Days Meeting_Day;
var int Cars_To_Sell[Days] in 0..20;
var float Distance_To_Drive;
var int+ Number_Of_Employees;
```

It is possible to declare a variable of the type `int` or `float` without further restricting them.
The first two examples restrict the values of the variable by specifying an interval after the variable name.
If no restriction is given (of course, potentially restricted by the normal constraints) then the variable can take on values from the entire `int`/ `float` domain, which is indeed very large. This declaration is used in the last two examples while the very last example restricts the values to be nonnegative. The data types `int+` and `float+` should be used whenever possible because they reduce the search space drastically.
Consider a situation where one knows that the desired values are definitely positive (the price for a product, the distances from factories to stores, etc.). When this is not specified, negative values will also be assigned to the variables and the solver wastes precious computational time. Even worse, the solver may find feasible solutions of, in worst case, an optimal solution with negative values. Obviously this solution is infeasible and therefore the model is inconsistent according to the problem formulation.
The model could be made consistent by adding the constraints, that these variables must be greater or equal 0. But after every assignment of these variables the constraint is checked and this means computational overload.
It should be said that it is very effective to tighten variables as best as possible. It reduces inconsistencies and unnecessary computational overhead.

OPL provides several reflexive functions , which are used to obtain information about the current state of the computation. This means that it is possible to get information about the current domain of a variable. The argument of a reflexive function is in general a variable. Especially in the search section of a model these reflexive functions are used.
A list of some reflexive functions follows:

- `dmin(int c)`: the minimum value in `dom(c)`
- `dmax(int c)`: the maximum value in `dom(c)`
- `dsize(int c)`: the size of `dom(c)`
- `bound(int i)`: 1 if `dmin(c) = dmax(c)` and 0 otherwise
- `bound(enum e)`: 1 if `dsize(e) = 1` and 0 otherwise.
- `dnexthigher(int c, int j)`: the smallest value greater than `j` in `dom(c)` or `j` if none exists

All functions but the last can also be used for floats.
These functions are very useful for modifying the search. Their use is explained in Section 1.4.

1.2.4 Example of a simple model

Imagine a gas production plant that produces ammonia gas (NH_3) and ammonium chloride (NH_4Cl). The company has 50 units of nitrogen (N), 40 units of chlorine (Cl) and

180 units of Hydrogen (H) at its disposal. There is a profit of 40 EUR for each ammonia gas unit and 50 EUR for each ammonium chloride unit.
The goal is to achieve the best profit for the stock of chemicals.

```
var float+ gas;
var float+ chloride;

maximize
  40 * gas + 50 * chloride
subject to {
  gas + chloride <= 50;
  3 * gas + 4 * chloride <= 180;
  chloride <= 40;
};
```

1.3 Stating Constraints

The constraint store allows the user to state all constraints of the model. The clear separation of the constraint store from the model also helps to produce more readable code.
There are different kinds of constraints like higher-order constraints and global constraints. The concepts behind these constraints will be explained in this section and furthermore we will see examples of how to state these constraints in OPL.
In this section we will introduce the different discrete constraints. OPL supports float constraints but they have to be linear or piecewise linear. The implemented algorithms for solving real linear constraints can in general not be applied to nonlinear problems. Additionally the operators `<,>` and `<>` are not allowed for float constraints. Therefore we will concentrate on discrete constraints for the further explanations.

1.3.1 Basic Constraints

Basic constraints are constructed from (discrete) constants, variables and arithmetic operators. OPL uses these constraints to reduce the domain of involved variables by applying various local consistency algorithms. This is not further explained in this paper. For more information about this topic reference the thesis of the author [2].
Code example:

```
var int freq[Freqs] in 0..256;
solve {
  forall(f,g in Freqs)
    abs(freq[f] - freq[g]) > 16;
};
```

The constraint ensures that all assigned frequencies have a certain distance from each other (in this example the distance is 16). This is crucial to prevent overlapping of frequencies.

1.3.2 Logical Combination of Constraints

Basic constraints can be combined with logical operators. In case of a logical "OR" (\/) like in the example below, the constraint ensures that at least one of the conditions is satisfied.
Code example:

```
var int freq[Freqs] in 0..256;
solve {
  forall(f,g in Freqs)
    abs(freq[f] - freq[g]) > 16 \/ freq[f] = freq[g];
};
```

1.3.3 Higher-order Constraints

Constraints can include other constraints as a part of their expressions. These embedded constraints are associated with a 0-1 integer variable that becomes 1 when the constraint can be satisfied and 0 otherwise. (see Page 10).
Code example:

```
range Range 0..50;
range Size 1..100;
var Range box[Size];
solve {
  sum(j in Size) (box[j] = 2) >= 3;
};
```

This constraint is satisfied if the array box contains at least three occurrences of the value 2.

1.3.4 Global Constraints

Global constraints are highly sophisticated constraints. They are implemented constraints acting as a black box. They are mostly problem-tailored and accessible through a keyword like `alldifferent`.
Global constraints handle specific algorithmic problems or subploblems in a more efficient way than one could by simply stating the problem. These typical algorithmic problems are for example the TSP (Travelling Salesperson Problem) or permutation problems. Global constraints are more efficient because a specific problem provides more information about its input and output. In the case of the `alldifferent(a)` constraint, which means that all elements in the array `a` have to have different values, one would need a lot of constraints to state the problem:

```
forall (i,j in range_of_array_a)
  a[i] <> a[j];
```

Checking whether all constraints are satisfied would be time consuming and the `alldifferent` constraint handles this faster.

A closer look at these constraints is not possible. They can be compared which functions or procedures in programming languages like C.
A list of some supported global constraints follows:

- `alldifferent`: Like mentioned above, it assures that all elements in an array have different values
- `circuit`: This constraint expects an array, indexed with the range 1..n, of integers with domains of the range 1..n. Each value in the range corresponds to a node in a graph and the value of an element in the array corresponds to the unique successor of the node.

  `circuit` holds if the so defined graph is a Hamiltonian circuit. Operationally the constraint removes values that would produce a subtour that is not a Hamiltonian circuit.
- `distribute`: This global constraint expects three arrays as arguments:

  `distribute(card,value,base)` where card and value must have the same index set.

  The constraint holds when `card[i]` is the number of occurences of `value[i]` in the array `base`.

  If the index set of `card` and `value` is `S` and the index set of `base` is `R`, then the constraint above is equivalent to, but more efficient than the constraint:

```
forall(i in S)
  card[i] = sum(j in R) (value[i] = base[j]);
```

1.4 Syntax for modifying the search strategies

OPL supports a great variety of tools to alter the default search strategy.
Following is a list of the instructions and keywords:

- `try/tryall`: Can be used to alter the order in which variables are instantiated and also the order of the assigned values.
- `if/then/else`: With the if-then-else statement one can make different decisions depending on certain circumstances.
- `while`: The same feature like in C for instance.
- `select`: Selects data for which certain circumstances hold.
- `once`: Searches for a feasible solution/assignment for its argument
- data-driven constructs: Execute an instruction whenever some condition becomes true or some event occurs.
- generation procedure: Assigns values to its variables.

These strategies are used when specific knowledge about the solution is at hand.
One might wish to assign a specific value to a variable at the beginning of the constraint propagation process and would do this with the `try`-instruction, for instance.

1.4.1 try/tryall

The `try` instruction specifies the order of some or all variables and initializes them with a specific value given in the instruction. The `try` command therefore is a tool to guide the variable and value ordering.
This strategy promises a speed up if the user has some specific knowledge about the solution. Say there is a variable "production" with a domain of 1 to 10 and it is more likely that in an optimal solution the value is 5 to 7.
The instruction is this case would be:

```
try
    production = 5
  |
    production = 6
  |
    production = 7
endtry;
```

When beginning to instantiate variables OPL would first instantiate the variable production with 5 and if the search fails (or if an optimal solution is searched) tries the value 6 and then 7. If the search fails with these values (or in case of optimization a better solution is searched) all other values in the domain will be considered to be assigned.
Technically speaking OPL adds the constraint `production = 5` to the constraint store. When the constraint is added OPL checks it for inconsistencies, tries the next value if there is one and then proceeds to the next instruction if successful.
The advantage of the `try` instruction is that if the suggested or supposed values fail, the different values in the domain will be assigned to the variable and aren't dropped. This means that even if the solution has other values for the variables than supposed, the search will not fail but only take more time due to wrong information.
An other important fact is that the search instructions are executed in the order they are written in the model.
Therefore one can control at what time a variable is to be examined and this means to control the position of the variable in the search tree. This can be static or dynamic. In the dynamic case, data-driven constructs check if a given condition holds and decide which value or variable to consider next.
Data-driven constructs will be introduced at a later point in this chapter.
Code example:

```
search {
when (dmax(production) = 5 do
  try production = 5
    |
      production = 4
  endtry;
```

In the example above the variable `production` is only instantiated when the number of values in its domain is exactly 5. When this is achieved (by propagation) then the variable will first be assigned the value 5 then, if it fails, the value 4. The line including the `when` keywords is the data-driven construct in this case.

The `tryall` instruction does mainly the same as `try` but is more convenient when the alternatives in a `try` instruction are too many to be written down exhaustively. `tryall` then is a compact representation of the `try` instruction.
For example:

```
tryall(i in 1..5)
  x = i;
```

is equivalent to the instruction

```
try x = 1 | x = 2 | x = 3 | x = 4 | x = 5 endtry;
```

The `tryall` instruction can also specify the order in which the values are assigned. This may be increasing, decreasing or by conditional choices.
Code example:

```
tryall(i in 1..10 ordered by decreasing i)
  x = i;

tryall(i in 1..10 : i mod 2 = 0)
  x  = i;
```

In the first example `i` is ordered from 10 to 1, and in the second example `i` is ordered 2,4,6,8,10. Namely all values that are even, which is the conditional choice in this example.

1.4.2 `if/then/else` Clause

In OPL it is possible to perform decision-based search. To do this we have the `if/then/else` clause. In the `then`-environment conditions can be specified that are only valid in case certain circumstances occur. This is very useful for example if some variables depend on the assigned values of other variables.
Code example:

```
search {
  forall(i in 1..10)
    if i mod 2 = 0 then
      tryall(v in 1..10 ordered by increasing v)
        x[i] = v

    else

      try(v in 1..10 ordered by decreasing v)
        x[i] = v
    endif;
}
```

1.4.3 `while` Instruction

With the `while` instruction one can execute a choice statement while a condition is satisfied.
Code example:

```
search {
  while not bound(x) do
    try x = dmin(x) | x <> dmin(x) endtry;
};
```

(See reflexive functions on side 12 for information about `bound`).

1.4.4 `select` Instruction

The `select` instruction selects data satisfying a certain condition. It is very useful in conjunction with the `while` instruction.
Code example:

```
search {
  while not bound(x) do
    select(i in 1..10 : not bound(x[i]))
      tryall(v in 1..10) x[i] = v;
};
```

In this example the variable `x` is assigned values as long as not all its array-fields are assigned.

1.4.5 `once` Statement

The `once` statement is used when only one solution is needed.
The instruction `once C` searches a solution for the constraint `C` and if succesful the solution is returned. This means that no backtracking occurs in `C` as soon as its first solution is found. In the example below the program terminates after the first solution is found. This means that no more attempts are made for further solutions.
Using the `once` statement in a wrong way can lead to inconsistencies and therefore may result in an erroneous termination of the program without a (feasible or infeasible) solution. See the second example in this example part. In that example the `tryall` instruction commits after its first solution (this adds the constaint `x = 0` to the constraint store). The constraint `x <> 0` then fails and no choice is left to explore. Therefore the program terminates without a solution.
The following example shows that using `once` in a wrong way leads to inconsistencies. In the first example `once` assigns a value to `x` and checks if it is not 0. When a feasible value is assigned the `once` environment is exited by the program correctly.
In the second example `once` assigns a value to `x` and exits immediatelly because there are no constraints for the assignment. As soon as the program leaves the `once` environment it checks if the assigned value is not 0. Since The assigned value is indeed 0 the program terminates without a solution, because it cannot backtrack into the `once` environment.
Code example:

```
var int x in 0..10;
search {
   once {
    tryall(i in 0..10)
      x = i;
    x <> 0;
  };
};

var int x in 0..10;
search {
   once {
    tryall(i in 0..10)
      x = i;
  };
   x <> 0;
};
```

1.4.6 Data-driven Constructs

The key idea of data-driven constructs is to execute an instruction whenever a certain condition becomes true or some event occurs. They can be viewed as processes that wait until some condition or event take place before being executed.
Two supported data-driven constructs are `when` and `onValue`.
The `when`-instruction waits until some conditions take places and then its body is executed.
The `onValue`-instruction waits until an expression has a fixed value to execute the body.
Code examples:

```
search{
  forall (i,j in 1..10)
    when array[i,j] = 0 do
      sum(k in 1..10) array[k,j] >= 40;
};

search{
  forall (i,j in 1..10)
    onValue array[i,j] do
      if array[i,j] = 0 then
         array[j,i] > 0
      endif;
};
```

In the first example the `when`-instruction is activated whenever an element `array1[i,j]` of the array `array1` evaluates to 0. Then the constraint ensures that the sum of the line `j` is at least 40.
In the second example the `onValue`-instruction is activated, when an element of `array1` is given a value. When this value is 0 the constraints ensures that the elements "counterpart"

is greater than 0.

1.4.7 Generation Procedures

The instruction `generate` receives a discrete variable or an array of discrete variables and generates a value or values for all these variables. The generation procedures are variable ordering procedures based on different variable ordering heuristics like "smallest domain first".
There are various generation instructions in OPL:

- `generateSize`: generates values for the variable with the smallest domain first
- `generationMin/generationMax`: generates values for the variable with the smallest/largest value in its domain first
- `generateSeq`: generates values for the variables in the fixed order of the array

The advantage of these kind of instructions is that the user can specify that she wants specific variables assigned but without using specific values. The different forms of the `generate` instruction are supports for different theoretical concepts like the "smallest-domain-first" strategy. For further informations about these strategies reference the thesis of the author [2]

Bibliography

[1] Pascal van Hentenryck: The OPL Optimization Programming Language. MIT Press, Cambridge, Mass.;

[2] Roland Martin: Modeling An Optimization Problem from the Automated Manufacturing of PC Boards. http://www.algo.tu-darmstadt.de/ martin/prints/thesis.ps