MiniZinc Tutorial

Ralph Becket
rafe@csse.unimelb.edu.au

18 June 2007

1 Modelling in MiniZinc

The G12 project defines three closely related modelling languages:

- **Zinc** is a very high level modelling language; because of its expressive power mapping Zinc models on to underlying solvers is an on-going research problem and existing language support is experimental. We do not discuss full Zinc further in this document.
- **MiniZinc** is a medium level modelling language that is comfortable to work in and for which we do have (very nearly) full support.
- **FlatZinc** is a low level modelling language intended primarily as input for compliant solvers which we hope will become a de facto standard.

The remainder of this document seeks to explain by example how to write and evaluate models in MiniZinc. In what follows, FD stands for "finite domain" and LP stands for "linear problem". FD solvers are used for integer problems while LP solvers are used for floating point problems.

1.1 Golomb Rulers (FD)

A Golomb ruler R is a set of integers such that all pairwise differences are different — that is, $\forall$ distinct $a, b, c, d \in R.(a-b) = (c-d) \Rightarrow (a, b) = (c, d)$.

Figure 1 shows a MiniZinc model of the problem (line numbers in the figure are there for reference and should not be included in the model file) and

```
 1 % Golomb rulers.
 2
 3 include "globals.mzn";
 4
 5 int: n = 4;
 6
 7 array [1..n] of var 0..(n * n): mark;
 8
 9 constraint mark[1] = 0;
10 constraint forall (i in 1..(n - 1)) (mark[i] < mark[i + 1]);
11 constraint
12     all_different (i in 1..n, j in (i + 1)..n) (mark[j] - mark[i]);
13
14 solve minimize mark[n];
15
16 output ["golomb: ", show(mark), "\n"];
```

Figure 1: MiniZinc model for Golomb Rulers.

the output from evaluating it with the `zinc` interpreter. Let's examine the model in more detail.

Line 1 is a comment. Comments begin with `%` and extend to the end of the line. Comments can occur anywhere in a model.

Line 3 includes the standard MiniZinc library file `globals.mzn`, which defines various global constraints (such as `all_different` used on line 12). All MiniZinc items that are not comments must be terminated with a semicolon.

Line 5 defines an integer parameter `n` with value `4`; this is the number of marks we wish our ruler to have.

Line 7 declares an array `mark` of `n` integer variables, indexed from `1` to `n`, each in the range `0..(n * n)`. Each array item corresponds to the location of a mark on our ruler.

Line 9 constrains the first mark to appear at position `0`.

Line 10 constrains the `mark` array items to appear in strictly ascending order.

Lines 11 and 12 use the `all_different` global constraint to ensure that the difference between each pair of distinct marks on the ruler is unique.

Line 14 specifies the problem goal: to find values for the `mark` array such that the last is minimized (the other possible solve goals are `solve maximize mark[n];` or `solve satisfy;` which looks for any solution).

Line 16 specifies how to print a solution if one is found. The function `show` converts its argument into a printable string. If a solution cannot be found then `No solution` will be printed instead. Omitting the output item will cause all of the variables in a model to be printed, one per line, if a solution is found.

In order to evaluate a model using the G12 implementation of MiniZinc, we must first convert that model into an equivalent FlatZinc model using the MiniZinc-to-FlatZinc converter, `mzn2fzn` .

Assuming that the model lives in the file `golomb.mzn` (note that all files that contain MiniZinc models must have the extension ".mzn"), we can convert it to FlatZinc as follows:

```
$ mzn2fzn golomb.fzn
```

If the model contains errors then `mzn2fzn` will report them, otherwise it will output a FlatZinc version of `golomb.mzn` in the file `golomb.fzn`. (FlatZinc files must have the extension ".fzn".)

We can then evaluate the model using a FlatZinc interpreter. The G12 FlatZinc interpreter is called `flatzinc` . It is invoked as follows:

```
$ flatzinc golomb.fzn
```

The output produced is:

```
[ 0, 1, 4, 6 ]
```

which, as required, has six unique differences: $\{1, 4, 6, 3, 5, 2\}$.

Syntactic sugar for arrays

The model uses convenient syntactic sugar for the `forall` and `all_different` constraints. In fact, both of these constraints take a single array argument. The form

```
forall (i in 1..(n - 1)) (mark[i] < mark[i + 1]);
```

is a more readable version of

```
forall([mark[i] < mark[i + 1] | i in 1..(n - 1)]);
```

(which uses an *array comprehension*) which in turn is short for

```
forall([mark[1] < mark[2], mark[2] < mark[3], mark[3] < mark[4]]);
```

Observe that array values are written as comma separated sequences of expressions, surrounded by brackets.

Similarly

```
all_different (i in 1..n, j in (i + 1)..n) (mark[j] - mark[i]);
```

is a more readable version of

```
all_different([mark[j] - mark[i] | i in 1..n, j in (i + 1)..n]);
```

which is short for

```
all_different([
    mark[2] - mark[1], mark[3] - mark[1], mark[4] - mark[1],
    mark[3] - mark[2], mark[2] - mark[2],
    mark[4] - mark[3]]);
```

You can use the special syntactic sugar in any expression where a function symbol is applied to a single array.

1.2 Sudoku (FD)

A Sudoku problem is a 3×3 grid of 3×3 subgrids. Each square must be assigned a number in the range 1..9 such that the numbers in each row, column, and subgrid are distinct.

Figure 1 shows a MiniZinc model of the problem (line numbers in the figure are there for reference and should not be included in the model file) and the output from evaluating it with the `zinc` interpreter. Let's examine the model in more detail.

```
% Sudoku solver.

include "globals.mzn";

array [1..9, 1..9] of var 1..9: sq;

predicate row_diff(int: r) =
    all_different (c in 1..9) (sq[r, c]);

predicate col_diff(int: c) =
    all_different (r in 1..9) (sq[r, c]);

predicate subgrid_diff(int: r, int: c) =
    all_different (i, j in 0..2) (sq[r + i, c + j]);

constraint forall (r in 1..9)         (row_diff(r));
constraint forall (c in 1..9)         (col_diff(c));
constraint forall (r, c in {1, 4, 7}) (subgrid_diff(r, c));

sq =
 [| _, _, _, _, _, _, _, _, _
  | _, 6, 8, 4, _, 1, _, 7, _
  | _, _, _, _, 8, 5, _, 3, _
  | _, 2, 6, 8, _, 9, _, 4, _
  | _, _, 7, _, _, _, 9, _, _
  | _, 5, _, 1, _, 6, 3, 2, _
  | _, 4, _, 6, 1, _, _, _, _
  | _, 3, _, 2, _, 7, 6, 9, _
  | _, _, _, _, _, _, _, _, _
 |];

solve satisfy;

output ["sq = ", show(sq), "\n"];
```

Figure 2: MiniZinc model for a Sudoku problem.

Line 3 includes `globals.mzn` which defines the `all_different` predicate.

Line 5 defines `sq`, a two dimensional array of squares, indexed by row and column number respectively, of integer variables in the range `1..9`.

Lines 7 and 8 define a predicate `row_diff(r)` which constrains the squares in row `r` to be distinct. **Lines 10 and 11** define `col_diff` for columns and **lines 13 and 14** define `subgrid_diff` for subgrids.

Lines 16 to 18 use the predicates to constrain the values in the puzzle. Observe that the range for index variables `r` and `c` in line 18 is given as the set `{1, 4, 7}` (set values are written as comma separated sequences of expressions, surrounded by braces) the expression `1..9` of course being shorthand for the set `{1, 2, 3, 4, 5, 6, 7, 8, 9}`.

Lines 20 to 30 constrains the value of `sq`, with the underscores denoting "don't know" values. Note that even though `sq` is a two-dimensional array, its value is given as a one-dimensional array in row-major order (literal MiniZinc array values like this are always one-dimensional).

Line 32 tells the solver to search for any solution and halt.

Running the model we get the following solution (which has been tidied up for visual effect):

```
$ mzn2fzn sudoku.mzn
$ flatzinc sudoku.fzn
sq =
  [ 5, 9, 3,  7, 6, 2,  8, 1, 4,
    2, 6, 8,  4, 3, 1,  5, 7, 9,
    7, 1, 4,  9, 8, 5,  2, 3, 6,

    3, 2, 6,  8, 5, 9,  1, 4, 7,
    1, 8, 7,  3, 2, 4,  9, 6, 5,
    4, 5, 9,  1, 7, 6,  3, 2, 8,

    9, 4, 2,  6, 1, 8,  7, 5, 3,
    8, 3, 5,  2, 4, 7,  6, 9, 1,
    6, 7, 1,  5, 9, 3,  4, 8, 2 ];
```

```
% Perfect squares.

int: n = 10;                    % Consider squares up to n*n.
array [0..n] of 0..n*n: sq = array1d(0..n, [y*y | y in 0..n]);
array [0..n] of var 0..n: s;    % Decreasing indices into sq.
var 0..n: x;                    % Compute sum equal to sq[x]
var 0..n: j;                    % from this many sub-squares.

constraint forall (i in 1..n) (s[i] > 0  ->  s[i - 1] > s[i]);
constraint s[0] < x;
constraint sum (i in 0..n) (sq[s[i]])  =  sq[x];
constraint s[j] > 0;

solve maximize j;

output [
    "x = ", show(x), "\n",
    "s = ", show(s), "\n",
];
```

Figure 3: MiniZinc model for computing squares equal to the sum of distinct smaller squares.

1.3 Perfect squares (FD with conditional constraints)

We can define a perfect square x^2 as one satisfying $x^2 = \sum \{y^2 \mid y \in S\}$ where S is a finite set of natural numbers smaller than x.

Figure 3 shows a MiniZinc model for computing perfect squares. Let's examine the model in more detail.

Line 3 fixes an upper bound `n` on the number of squares the model will consider.

Line 4 defines a constant array `sq` containing the squares from `0` up to `n*n` by means of an array comprehension.

Line 5 defines an array `s` of variables, each being an index into the array `sq` (we are going to compute $x^2 = \sum_{i \,\in\, \texttt{0..n}} \texttt{sq[s[}i\texttt{]]}$ — this is our encoding of the set S).

Line 6 defines a variable `x` which is the index of `sq` containing our perfect square (i.e., $x^2 =$`sq[x]`).

Line 7 defines a variable `j` which is the cardinality of our set S summing to `sq[x]`.

Line 9 constrains the array of indices `s` to be a strictly decreasing sequence followed by zeroes. The arrow `->` denotes logical implication.

Line 10 requires the set S to be non-empty.

Line 11 constrains the sum of the squares indexed by `s` to be equal to `sq[x]`.

Line 12 constrains the members of S to be non-zero (`sq[s[j]]` is the smallest member of S).

Line 14 requires the solver to find the largest set S given the bound `n`.

Evaluating the model we get:

```
x = 10
s = [ 7, 5, 4, 3, 1, 0, 0, 0, 0, 0, 0 ]
```

and indeed $10^2 = 7^2 + 5^2 + 4^2 + 3^2 + 1^2$.

1.4 Production planning (LP)

Figure 4 shows a MiniZinc model for a production planning problem (line numbers in the figure are there for reference and should not be included in the model file). In this problem a company manufactures three kinds of products (e.g., kinds of pasta) using two kinds of resources (e.g., eggs and milk). In-house production of each product is limited by the resources in stock. The model must decide how much of each product to make in-house versus how much to outsource in order to meet expected demand and minimize costs.

Let's examine the model in more detail.

Lines 3 and 4 define two integer sets giving symbolic names to each product and resource kind.

Lines 7 to 9 define arrays giving the expected demand, the in-house production cost, and the outsourcing production cost for each product. Note that we can use set expressions (`products` in this case) as the index sets for arrays. Note also that in MiniZinc floating point constants must include a fractional part: `42` is an `int`; `42.0` is a `float`.

Line 12 defines a two-dimensional array, indexed by product and resource kind respectively, giving the resource requirements for in-house manufacturing.

Line 17 defines an array giving the available stocks of each resource kind.

Lines 20 and 21 define arrays of problem variables for the amount of each product to manufacture in-house and outsource respectively.

Lines 24 and 25 constrain the problem variables to be non-negative.

Lines 28 to 32 state that in-house production cannot use more than the available resource stocks.

Lines 34 to 37 require that manufacturing plus outsourcing must meet demand for each product.

Lines 40 to 42 specifies the objective function, namely to minimize the cost of meeting the expected demand.

```
% Simple production planning example adapted from the OPL book.

set of int: products = 1..3;   % We make 3 products.
set of int: resources = 1..2;  % We use 2 kinds of resources.

% Demand, in-house cost, and outsourcing cost for each product:
array[products] of float: demand   = [100.0, 200.0, 300.0];
array[products] of float: in_cost  = [  0.6,   0.8,   0.3];
array[products] of float: out_cost = [  0.8,   0.9,   0.4];

% Resources of each kind needed to manufacture each product unit:
array[products, resources] of float: consumption = [|0.5, 0.2,
                                                    |0.4, 0.4,
                                                    |0.3, 0.6|];

% Current in-house stocks of each resource kind:
array[resources] of float: capacity = [20.0, 40.0];

% Variables: how much should be made in-house or outsourced:
array[products] of var float: in_house;
array[products] of var float: outsource;

% Production cannot be negative:
constraint forall (p in products) (in_house[p]  >= 0.0);
constraint forall (p in products) (outsource[p] >= 0.0);

% In-house production cannot use more than resource stocks:
constraint forall (r in resources) (
    sum (p in products) (consumption[p, r] * in_house[p]) <= capacity[r]
);

% Production must meet demand:
constraint forall (p in products) (
      in_house[p] + outsource[p] >= demand[p]
);

% Goal: minimize costs:
solve minimize sum (p in products) (
      in_cost[p] * in_house[p]  +  out_cost[p] * outsource[p]
);

output [
    "in_house  = ", show(in_house),  "\n",
    "outsource = ", show(outsource), "\n",
];
```

Figure 4: MiniZinc model for a production planning problem.

```
% Contrived problem.

include "globals.mzn";

var 0..9: x;
var 0..9: y;
var 0..9: z;

constraint all_different([x, y, z]);
constraint y - 3 < x  /\  x < y + 3;
constraint z - 3 < y  /\  y < z + 3;

solve satisfy;

output ["x = ", show(x), "; ",
        "y = ", show(y), "; ",
"z = ", show(z), "\n"];
```

Figure 5: Contrived MiniZinc model to illustrate search control.

Running the model we get

```
in_house  = [ 40.0, 0.0, 0.0 ]
outsource = [ 60.0, 200.0, 300.0 ]
```

1.5 Controlling search

Consider the contrived problem model in figure 5 which states that `x` must be within `3` of `y`, `y` must be within `3` of `z`, and all three variables must be distinct in the range `0..9` (observe the use of the logical conjunction operator `/\` which isn't strictly necessary — conjunctions can be split into separate constraints — but does improve readability here).

The default search procedure produces `x = 2; y = 0; z = 1` .

The search procedure can be changed in MiniZinc models by annotating the `solve` goal. For example, with:

```
solve
 :: int_search([x, y, z], first_fail, indomain_median, complete)
    satisfy;
```

we get `x = 5; y = 6; z = 7` .

The `int_search` annotation takes four arguments:
— an array of the variables affected by this annotation;
— a variable selection strategy;
— a variable domain reduction strategy; and
— a search strategy
respectively.

In the example annotation the `first_fail` variable selection strategy always chooses the undecided variable with the smallest domain, the `indomain_median` nondeterministically splits that variable's domain in half (so, for example, `0..9` reduces to `0..4` on one search branch and `5..9` on another), and the `complete` strategy guarantees not to omit solutions from the search space.

Multiple annotations can appear, although the order in which they are carried out is not defined. For example, with:

```
solve
 :: int_search([x], first_fail, indomain_median, complete)
 :: int_search([y, z], smallest, indomain_max, complete)
    satisfy;
```

we may get either `x = 4; x = 6; z = 8` or `x = 7; y = 9; z = 8`.

We can force the search annotations to be carried out in order by using a `seq_search` annotation, as in the following:

```
solve
  :: seq_search([
       int_search([x], first_fail, indomain_median, complete),
       int_search([y, z], smallest, indomain_max, complete)])
  satisfy;
```

Other search annotations, `float_search`, `bool_search`, and `set_search` are available for float, bool and set variables respectively.

Currently supported variable selection strategies for `int_search`

The current implementation of `flatzinc` supports the following variable selection strategies for `int_search`:

- `first_fail` which chooses the (undecided) variable with the smallest domain;
- `anti_first_fail` which chooses the variable with the largest domain;
- `smallest` which chooses the variable with the smallest lower bound;
- `largest` which chooses the variable with the largest upper bound; and
- `input_order` which chooses variables in the order specified in the array.
bound;

Currently supported domain reduction strategies for `int_search`

The following domain reduction strategies are supported:
- `indomain` which reduces a domain to a singleton value in an arbitrary order;
- `indomain_min` which reduces a domain to a singleton value in an ascending order;
- `indomain_max` which reduces a domain to a singleton value in a descending order;
- `indomain_median` which splits a domain in half; and
- `indomain_split` which splits a domain about the mean of its lower and upper bounds.

Currently supported search strategies for `int_search`

Only the `complete` search strategy is currently supported.

2 Comments, suggestions, and bug reports

Bugs can be reported via the G12 bug tracking system at `bugs.g12.csse.unimelb.edu.au`.

Comments, questions and suggestions should be sent to the G12 Users mailing list. You can subscribe to the list by sending an e-mail containing the word `subscribe` in the body to `g12-users-request@csse.unimelb.edu.au`. Thereafter, mail may be sent to `g12-users@csse.unimelb.edu.au`.