

Constraint Programming

Introduction to the MiniZinc Modelling Language

Alan M. Frisch

Artificial Intelligence Group
Department of Computer Science
University of York

Last updated: 18 Oct 2013

SEND-MORE-MONEY Problem

- Assign a unique digit to each letter such that the following sum holds

$$\begin{array}{rcccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

- It is understood that S and M cannot be 0 since leading zeroes are usually not written.

MiniZinc Model for SEND-MORE-MONEY

Adapted Hakan Kjellerstrand: www.hakank.org/weblogg/archives/001209.html

```
include "globals.mzn";
```

include item

```
var 0..9: S;
```

```
var 0..9: E;
```

```
var 0..9: N;
```

```
var 0..9: D;
```

```
var 0..9: M;
```

```
var 0..9: O;
```

```
var 0..9: R;
```

```
var 0..9: Y;
```

variable items

3 other kinds of items:

- assignment items
- predicate and test items
- annotation items

- Order of items is irrelevant
- Can have multiple items of same kind

```
constraint
```

```
  all_different([S,E,N,D,M,O,R,Y]) /\
```

```
      1000*S + 100*E + 10*N + D
```

```
      1000*M + 100*O + 10*R + E
```

```
10000*M + 1000*O + 100*N + 10*E + Y
```

```
 /\
```

```
 S > 0 /\
```

```
 M > 0;
```

+

=

constraint item

```
solve satisfy;
```

```
output [ "      ", show(S), show(E), show(N), show(D), "\n",
```

```
        " + ", show(M), show(O), show(R), show(E), "\n",
```

```
        " = ", show(M), show(O), show(N), show(E), show(Y), "\n"
```

```
];
```

solve item

output item

MiniZinc Model for SEND-MORE-MONEY

Some Remarks

```
constraint
  all_different([S,E,N,D,M,O,R,Y]) /\
    1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E =
  10000*M + 1000*O + 100*N + 10*E + Y
  /\
  S > 0 /\
  M > 0;
```

Could be written as

```
constraint all_different([S,E,N,D,M,O,R,Y]);

constraint
  1000*S + 100*E + 10*N + D +
  1000*M + 100*O + 10*R + E =
  10000*M + 1000*O + 100*N + 10*E + Y;

constraint S > 0;

constraint M > 0;
```

MiniZinc Model for SEND-MORE-MONEY

Some Remarks

```
constraint all_different([S,E,N,D,M,O,R,Y]);
```

Could be written as

```
array[1..8] of var int : fd;
```

variable item

```
fd = [S,E,N,D,M,O,R,Y];
```

assignment item

```
constraint all_different(fd);
```

Or as

```
array[1..8] of var int : fd = [S,E,N,D,M,O,R,Y];
```

variable item

```
constraint all_different(fd);
```

MiniZinc Model for 8-Queens

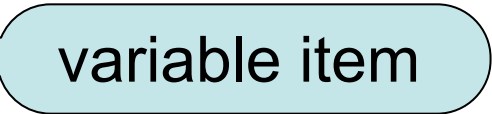
```
include "globals.mzn";
array [1..8] of var 1..8: q;
constraint all_different(q);
constraint forall (i in 1..8, j in i+1..8)(
    q[i] + 1 != q[j] + j /\
    q[i] - 1 != q[j] - j);

solve satisfy;
output [.....]
```

Better yet, use a parameter instead of 8

```
include "globals.mzn";
par int: n = 8;
array [1..n] of var 1..n: q;
constraint all_different(q);
constraint forall (i in 1..n, j in i+1..n)(
    q[i] + 1 != q[j] + j /\
    q[i] - 1 != q[j] - j);

solve satisfy;
output [.....]
```



MiniZinc Model for n-Queens

```
include "globals.mzn";  
par int: n;  
array [1..n] of var 1..n: q;  
constraint all_different(q);  
constraint forall (i in 1..n, j in i+1..n) (  
    q[i] + 1 != q[j] + j /\  
    q[i] - 1 != q[j] - j);  
solve satisfy;  
output [.....
```

value of n not given

queens.mzn

```
n=8;
```

queens.dzn

To run use the command

```
minizinc queens.mzn queens.dzn
```

or, equivalently

```
minizinc queens.mzn -D "n=8"
```

MiniZinc Model for n-Queens

Some Remarks

```
include "globals.mzn";  
par int: n;  
array [1..n] of var 1..n: q;  
constraint all_different(q);  
constraint forall (i in 1..n, j in i+1..n) (  
    q[i] + i != q[j] + j /\  
    q[i] - i != q[j] - j);  
solve satisfy;  
output [.....
```

It is an “array of var” not a “var array”

par int: n
can be abbreviated to
int: n;

MiniZinc Model for n-Queens with Optimization

Instead of finding a satisfiable solution, we could find a maximal one

```
include "globals.mzn";
par int: n;
array [1..n] of var 1..n: q;
constraint all_different(q);
constraint forall (i in 1..n, j in i+1..n)(
    q[i] + 1 != q[j] + j /\
    q[i] - 1 != q[j] - j);
solve maximize q[n];
output [.....
```

Or we could find a minimal one.

```
    :
    :
solve minimize q[n];
    :
    :
```

The objective can be any arithmetic (type int or float) expression

Parameters

In MiniZinc there are two kinds of variables:

Parameters are like variables in a standard programming language. They must be assigned a value (but only once), and this is done during the compilation stage (not during solving process).

They are declared with a type (or a range/set).

You can use `par` but this is optional

The following are equivalent

```
int: i=3;
```

```
par int: i=3;
```

```
int: i;      i=3;
```

Decision Variables

Decision variables are like variables in mathematics. They are declared with a type and the `var` keyword. Their value is computed by MiniZinc so that they satisfy the model

Typically they are declared using a **range** or a **set** rather than a type name

The range or set gives the domain for the variable.

The following are equivalent

```
var int: i; constraint i >= 0; constraint i <= 4;  
var 0..4: i;  
var {0,1,2,3,4}: i;
```

Question: what does this mean `constraint i = i + 1;`

Types

Available types are

- Integer: **int** or range **1..n** or set of integers
- Floating point number: **float** or range **1.0..f** or set of floats
- Boolean: **bool**
- Strings: **string** (but these cannot be decision variables)
- Arrays whose elements are of any type except arrays. The elements can be decision variables.
- Sets whose elements are of a primitive type.

Instantiations

Variables have an **instantiation** which specifies if they are parameters or decision variables.

The type + instantiation is called the type-inst.

MiniZinc errors are often couched in terms of mismatched type-insts...

Lists

Descriptions of MiniZinc often use the word “list.”

“List” is just an abbreviation used in describing MiniZinc.

A list is an array that is indexed by $1..n$ for some n .

There is nothing that the Minizinc language itself calls a list.

`list` is not a keyword.

Strings

Strings are provided for output

- An output item has form
 output <list of strings>;
- String literals are like those in C: enclosed in “ ”
- They cannot extend across more than one line
- Backslash for special characters \n \t etc
- Built in functions are
 - show(v)
 - “house”++“boat” for string concatenation

Arithmetic Expressions

MiniZinc provides the standard arithmetic operations

- Floats: `*` `/` `+` `-`
- Integers: `*` `div` `mod` `+` `-`

Integer and float literals are like those in C

There is no automatic coercion from integers to floats

The built-in `int2float(intexp)` must be used to explicitly coerce them

The arithmetic relational operators are

`==` `!=` `>` `<` `>=` `<=` (also `=` is same as `==`)

Basic Structure of a Model

A MiniZinc model is a sequence of items

The order of items does not matter

The kinds of items are

- An inclusion item

 - include <filename (which is a string literal)>;

- An output item

 - output <list of string expressions>;

- A variable declaration

- A variable assignment

- A constraint

 - constraint <Boolean expression>;

Basic Structure of a Model

The kinds of items (cont.)

- A solve item (a model must have exactly one of these)
 - solve satisfy;
 - solve maximize <arith. expression>;
 - solve minimize <arith. expression>;
 - Predicate and test items: macros of type **bool**
 - Annotation items
-
- Identifiers in MiniZinc start with a letter followed by other letters, underscores or digits
 - In addition, the underscore ``_`` is the name for an anonymous decision variable
 - example: `[_,_,_]` is a list of three un-named variables.

Arrays

An array can be multi-dimensional. It is declared by

array[*index_set 1*,*index_set 2*, ...,] **of type**

The index set of an array needs to be

a fixed integer range (contiguous), or

a fixed set expression whose value is an integer range (contiguous).

The elements in an array can be anything except another array.

They can be decision variables. For example

```
array[products, resources] of int: consumption;
```

```
    % array of parameters
```

```
array[products] of var 0..mproducts: produce;
```

```
    % array of variables
```

The built-in function **length** returns the number of elements in a 1-D array.

Arrays (Cont.)

1-D arrays are initialized using a list

```
profit = [400, 450];  
capacity = [4000, 6, 2000, 500, 500];
```

2-D array initialization uses a list with ``|'' separating rows

```
consumption= [| 250, 2, 75, 100, 0,  
               | 200, 0, 150, 150, 75 |];
```

Arrays of *any* dimension ($well \leq 3$) can be initialized from a list using the `arraynd` family of functions:

```
consumption= array2d(1..2,1..5,  
                    [250,2,75,100,0,200,0,150,150,75]);
```

The concatenation operator `++` can be used with 1-D arrays:

```
profit = [400]++[450];
```

Sets

Sets are declared by
set of type

They are only allowed to contain integers, floats or Booleans.

Set expressions:

- Set literals are of form $\{e_1, \dots, e_n\}$

- Integer or float ranges are also sets

- Standard set operators are provided:

 - in, union, intersect, subset, superset, diff, symdiff

- The size of the set is given by card

Some examples:

- set of int: `products = 1..nproducts;`

- `{1,2} union {3,4}`

Set par variable names, set literals or ranges can be used as types.

Array & Set Comprehensions

MiniZinc provides *comprehensions* (similar to ML)

A set comprehension has form

$\{ \text{expr} \mid \text{generator 1, generator 2, ...} \}$

$\{ \text{expr} \mid \text{generator 1, generator 2, ... where bool-exp} \}$

An array comprehension is similar

$[\text{expr} \mid \text{generator 1, generator 2, ...}]$

$[\text{expr} \mid \text{generator 1, generator 2, ... where bool-exp}]$

Some examples

$\{ i + j \mid i, j \text{ in } 1..3 \text{ where } i < j \}$
= $\{ 1 + 2, 1 + 3, 2 + 3 \}$
= $\{ 3, 4, 5 \}$

Array & Set Comprehensions

Exercise: What does b equal?

```
set of int: cols = 1..5;
set of int: rows = 1..2;
array[rows,cols] of int: a=[ | 250, 2, 75, 100, 0,
                             | 200, 0, 150, 150, 75| ];
b = array2d(cols, rows, [a[i,j] | j in cols,
                             i in rows]);
```

Iteration/Aggregation

MiniZinc provides a variety of built-in functions for iterating over a list or set:

- List of numerical expressions: sum, product, min, max
- List of constraints (Boolean expressions): forall, exists

MiniZinc provides a special syntax for calls to these (and other **generator** functions)

For example,

```
forall (i, j in 1..10 where i < j) (a[i] != a[j]);
```

is equivalent to

```
forall ([a[i] != a[j] | i, j in 1..10 where i < j]);
```


Assertions

Defensive programming requires that we check that the data values are **valid**.

The built-in Boolean function `assert(boolexp,stringexp)` is designed for this. It returns true if `boolexp` holds, otherwise prints `stringexp` and aborts

Like any other Boolean expression it can be used in a constraint item

For example,

```
int: nresources;  
constraint assert(nresources > 0,  
                  "Error: nresources =< 0");  
  
array[resources] of int: capacity;  
constraint assert forall(r in resources)(capacity[r] >= 0),  
                  "Error: negative capacity");
```

Exercise: Write an expression to ensure consumption is non-negative

```
array[products, resources] of int: consumption;
```

Assertions for Debugging

- You can (ab)use assertions to help debug

- `int: n = 5;`

- `array[1..n] of var 1..n: a;`

- `array[1..n] of 1..n: b = [3,5,2,3,1];`

- `constraint forall(j in 1..n, i in b[n-j]..b[n-1])
 (a[j] < i);`

- Error message

error:

debug.mzn:5

In constraint.

In 'forall' expression.

In comprehension.

`j = 5`

In comprehension head.

In '..' expression

In array access.

In index argument 1

Index out of range.

An array index is out of range
when j is 5.

Which one?

Let's check if it is b[n-j], that is
whether b-j is in 1..n

Assertions for Debugging

- You can (ab)use assertions to help debug

- `int: n = 5;`

- `array[1..n] of var 1..n: a;`

- `array[1..n] of 1..n: b = [3,5,2,3,1];`

- `constraint forall(j in 1..n)(`

- `assert(n-j in 1..n, "b[" ++ show(n-j) ++ "]");`

- Error message

error:

debug.mzn:6

In constraint.

In 'forall' expression.

In comprehension.

j = 5

In comprehension head.

In 'assert' expression.

Assertion failure: "b[0]"

So here we check
whether n-j in 1..n.

If-then-else

- MiniZinc provides an
if <boolexp> then <exp> else <exp> endif
expression
- For example,
if y != 0 then x / y else 0 endif
- The Boolean expression is not allowed to contain decision variables, only parameters
- In output items the built-in function **fix** checks that the value of a decision variable is fixed and coerces the instantiation from decision variable to parameter.

Constraints

- Constraints are the core of the MiniZinc model
- We have seen simple relational expressions but constraints can be considerably more powerful than this.
- A constraint is allowed to be any Boolean expression
- The Boolean literals are
true and false
and the Boolean operators are
 \wedge \vee \leftarrow \rightarrow \leftrightarrow not
- Global constraints: alldifferent

Complex Constraint Example

Imagine a scheduling problem in which we have a set of tasks that use the same single resource

Let $\text{start}[i]$ and $\text{duration}[i]$ give the start time and duration of task i

To ensure that the tasks do not overlap

```
constraint forall (i,j in tasks where i != j) (  
    start[i] + duration[i] <= start[j]  \/   
    start[j] + duration[j] <= start[i] );
```

Array Constraints

Recall that array access is given by $a[i]$.

The index i is allowed to be an expression involving decision variables in which case it is an implicit constraint on the array.

As an example consider the [stable marriage problem](#).

We have n (straight) women and n (straight) men.

Each man has a ranked list of women and vice versa

We want to find a husband/wife for each women/man s.t all marriages are stable, i.e.,

- Whenever m prefers another women o to his wife w , o prefers her husband to m
- Whenever w prefers another man o to her husband m , o prefers his wife to w

Stable Marriage Problem

```
int: n;  
array[1..n,1..n] of int: rankWomen;  
array[1..n,1..n] of int: rankMen;  
  
array[1..n] of var 1..n: wife;  
array[1..n] of var 1..n: husband;  
  
constraint forall (m in 1..n) (husband[wife[m]] = m);  
constraint forall (w in 1..n) (wife[husband[w]] = w);
```

Exercise: insert stability constraints here...

```
solve satisfy;  
output ["wives= ", show(wife), "\n",  
        "husbands= ", show(husband)];
```


Higher-order constraints

- The built-in coercion function `bool2int` allows the modeller to use so called `higher order` constraints:
- `Magic series problem`: find a list of numbers $S = [s_0, \dots, s_{n-1}]$ s.t. s_i is the number of occurrences of i in S .
- A MiniZinc model is

```
int: n;  
array[0..n-1] of var 0..n: s;  
  
constraint  
  forall(i in 0..n-1) (  
    s[i] = sum(j in 0..n-1) (bool2int(s[j]=i));  
  
solve satisfy;
```

Set Constraints

- MiniZinc allows sets over integers to be decision variables
- Consider the 0/1 knapsack problem

```
int: n;  
int: capacity;  
  
array[1..n] of int: profits;  
array[1..n] of int: weights;  
  
var set of 1..n: knapsack;  
  
constraint sum (i in knapsack) (weights[i]) <= capacity;  
  
solve maximize sum (i in knapsack) (profits[i]) ;  
  
output [show(knapsack)];
```

Set Constraints (Cont.)

- But this doesn't work—we can't iterate over variable sets
- **Exercise:** Rewrite the example so that it doesn't iterate over a var set

Enumerated Types

- Enumerated types are useful to name classes of object which we will decide about. In reality they are placeholders for integers
- `enum people = { bob, ted, carol, alice };`
- This can be imitated by
- ```
set of int: people = 1..4;
 int: bob = 1;
 int: ted = 2;
 int: carol = 3;
 int: alice = 4;
 array[people] of string: name =
 ["bob", "ted", "carol", "alice"];
```

# How MiniZinc work?

**minizinc** is a script.

- 1) It interprets the model and data and outputs a simpler form of model: FlatZinc
  - The tool `mzn2fzn` explicitly does this step.
  - Use the command `mzn2fzn file.mzn data.dzn`
    - creates `file.fzn`
- 2) FlatZinc interpreters/solvers run FlatZinc files
  - very simple output (just some variable values)
- 3) MiniZinc reads the simple output and calculates the complex output

# Running MiniZinc

- `minizinc [options] <model>.mzn [<file>.dzn ... ]`
  - appears -D option can come at end
  - I don't know what solver it uses by default (guess: G12 finite domain)

Alternative commands with same syntax and switches

- `mzn-g12fd`            uses G12 finite domain solver
- `mzn-g12cpx`            uses G12 CPX solver, which integrates a solver to handle explanations
- `mzn-g12lazy`           predecessor to CPX solver. Now deprecated.
- `mzn-g12mip`            uses a mixed-integer programming solver
- `mzn-g12sat`            uses a Boolean SAT solve

# Running MiniZinc

There are also solvers available from people outside the G12.

- **Opturion CPX**: a commercial version of the G12 CPX system. Might be fast and reliable. Free for academic use.  
<http://www.opturion.com/cpx.html>
- **Gecode**: a constraint library for C
- **Eclipse**: a constraint logic programming language
- **SiCStus Prolog**: a constraint logic programming language
- **JaCoP**: a constraint library for JAVA
- **SCIP**: a solver based on constraint integer programming
- **fzn2smt**: translates to SMTlib standard for SMT (SAT modulo theories) solvers

See MiniZinc README file for URLs to these other solvers

Not all features of the language are supported by all solvers.