FlatZinc Summary
Version 1.0

Ralph Becket

1 Introduction

This document is intended as a concise description of the FlatZinc modelling language. The reader should consult the "Specification of FlatZinc" for an authoritative reference. Throughout this document: r_1, r_2 denote float literals; x_1, x_2, $\ldots x_k$, x_i, n, i, j, k denote int literals; y_1, y_2, $\ldots y_k$, y_i denote literal array elements.

2 Types

2.1 Parameter types

Parameters are fixed quantities in the model.

`bool`	— `true` or `false`
`float`	— unbounded float
r_1`..`r_2	— bounded float
`int`	— unbounded int
x_1`..`x_2	— int in range
$\{x_1$`, `x_2`, ..., `$x_k\}$	— int in set
`set of int`	— subset of int
`set of` x_1`..`x_2	— subset of int range
`set of` $\{x_1$`, `x_2`, ...`$x_k\}$	— subset of int set
`array [1..`n`] of bool`	— array of bools
`array [1..`n`] of float`	— array of unbounded floats
`array [1..`n`] of` r_1`..`r_2	— array of floats in range
`array [1..`n`] of int`	— array of unbounded ints
`array [1..`n`] of` x_1`..`x_2	— array of ints in range
`array [1..`n`] of set of int`	— array of sets of ints
`array [1..`n`] of set of` x_1`..`x_2	— array of sets of ints in range
`array [1..`n`] of set of` $\{x_1$`, `x_2`, ...`$x_k\}$	— array of subsets of set

A range x_1`..`x_2 denotes a closed interval $\{x | x_1 \leq x \leq x_2\}$.

An array type appearing in a predicate declaration may use just `int` instead of `1..`n for the array index range in cases where the array argument can be of any length.

2.2 Variable types

Variables are quantities decided by the solver.

```
var bool
var float
var r1..r2
var int
var x1..x2
var {x1, x2, ..., xk}
var set of x1..x2
var set of {x1, x2, ..., xk}
array [1..n] of var bool
array [1..n] of var float
array [1..n] of var r1..r2
array [1..n] of var int
array [1..n] of var x1..x2
array [1..n] of var set of x1..x2
array [1..n] of var set of {x1, x2, ...xk}
```

An array type appearing in a predicate declaration may use just `int` instead of `1..`n for the array index range in cases where the array argument can be of any length.

2.3 Literal values

Examples of literal values:

Type	**Literals**
`bool`	`true, false`
`float`	`2.718`, `-1.0`, `3.0e8`
`int`	`-42`, `0`, `69`
`set of int`	`{}`, `{2, 3, 5}`, `1..10`
`arrays`	`[]`, `[`y_1`, ...`y_k`]`

where each array element y_i is either: a non-array literal; the name of a non-array parameter or variable, v; or a subscripted array parameter or variable, $v[j]$, where j is an int literal.

3 FlatZinc models

A FlatZinc model consists of:

1. zero or more external predicate declarations;
2. zero or more parameter declarations;
3. zero or more variable declarations;
4. zero or more constraints;

5. a solve goal

in that order.

FlatZinc syntax is insensitive to whitespace.

Comments start with a per cent sign, %, and extend to the end of the line. Comments can appear anywhere in a model.

3.1 Predicate declarations

Predicates used in the model that are not standard FlatZinc must be declared before any other lexical items at the top of a FlatZinc model. Predicate declarations take the form

```
predicate predname(type: argname, ...);
```

3.2 Parameter declarations

Parameters have fixed values and must be assigned values:

```
paramtype: paramname = literal;
```

3.3 Variable declarations

Variables have variable types and cannot have assigned values, except for arrays where an assignment is optional (it is often convenient to have fixed permutations of other variables). Variables may be declared with zero or more annotations.

```
vartype: varname [:: annotation]* [ = arrayliteral];
```

3.4 Constraints

Constraints take the following form and may include zero or more annotations:

```
constraint predname(arg, ...) [:: annotation]*;
```

where each argument argis either: a literal value; the name of a parameter or variable, v; or a subscripted array parameter or variable, $v[j]$, where j is an int literal.

3.5 Solve goal

A model should finish with a solve goal, taking one of the following forms:

```
solve [:: annotation]* satisfy;
```

or

solve [:: annotation]* minimize arg;

or

solve [:: annotation]* maximize arg;

3.6 Annotations

Annotations are optional suggestions to the solver concerning how individual variables and constraints should be handled (e.g., a particular solver may have multiple representations for int variables) and how search should proceed. An implementation is free to ignore any annotations it does not recognise, although it should print a warning on the standard error stream if it does so.

An annotation is either

annotationname

or

annotationname(annotationarg, ...)

where annotationarg is either a literal value or another annotation.

3.6.1 Search annotations

While an implementation is free to ignore any or all annotations in a model, it is recommended that implementations at least recognise the following standard annotations for solve goals.

seq_search([searchannotation, ...])

allows more than one search annotation to be specified in a particular order (otherwise annotations can be handled in any order).

A searchannotation is one of the following:

```
int_search(vars, varchoiceannotation, labellingannotation, strategyannotation)
bool_search(vars, varchoiceannotation, labellingannotation, strategyannotation)
set_search(vars, varchoiceannotation, labellingannotation, strategyannotation)
```

where vars is an array variable name or an array literal specifying the variables to be labelled (ints, bools, or sets respectively).

varchoiceannotation specifies how the next variable to be labelled is chosen at each choice point. Possible choices are as follows (it is recommended that implementations support the starred options):

`input_order`	⋆	Choose variables in the order they appear in vars.
`first_fail`	⋆	Choose the variable with the smallest domain.
`anti_first_fail`		Choose the variable with the largest domain.
`smallest`		Choose the variable with the smallest value in its domain.
`largest`		Choose the variable with the largest value in its domain.
`occurrence`		Choose the variable with the largest number of attached constraints.
`most_constrained`		Choose the variable with the smallest domain, breaking ties using the number of constraints.
`max_regret`		Choose the variable with the largest difference between the two smallest values in its domain.

labellingannotation specifies how the chosen variable should be labelled. Possible choices are as follows (it is recommended that implementations support the starred options):

`indomain_min`	⋆	Assign the smallest value in the variable's domain.
`indomain_max`	⋆	Assign the largest value in the variable's domain.
`indomain_middle`		Assign the value in the variable's domain closest to the mean of its current bounds.
`indomain_median`		Assign the middle value in the variable's domain.
`indomain_random`		Assign a random value from the variable's domain.
`indomain_split`		Bisect the variable's domain, excluding the upper half first.
`indomain_reverse_split`		Bisect the variable's domain, excluding the lower half first.
`indomain_interval`		If the variables domain consists of several contiguous intervals, reduce the domain to just one of the intervals. Otherwise just split the variable's domain.

Of course, not all labelling strategies make sense for all search annotations (e.g., `bool_search` and `indomain_split`).

Finally, strategyannotation specifies a search strategy; implementations should at least support `complete` (i.e., exhaustive search).

3.6.2 Output annotations

All output should be sent to the standard output stream.

Model output is specified through variable annotations. Non-array output variables should be annotated with `output_var`. Array output variables should be annotated with `output_array([`x_1`..`x_2`, ...])` where x_1`..`x_2`, ...` are the index set ranges of the original array (it is assumed that the FlatZinc model was derived from a higher level model written in, say, MiniZinc, where the original array may have had multiple dimensions and/or index sets that do not start at 1).

3.6.3 Variable definition annotations

To support solvers capable of exploiting functional relationships, a variable defined as a function of other variables may be annotated thus:

```
var int: x :: is_defined_var;
```

```
...
constraint int_plus(y, z, x) :: defines_var(x);
```

(The `defines_var` annotation should appear on exactly one constraint.) This allows a solver to represent `x` internally as a representation of `y+z` rather than as a separate constrained variable. The `is_defined_var` annotation on the declaration of `x` provides "early warning" to the solver that such an option is available.

3.6.4 Intermediate variables

Intermediate variables introduced during conversion of a higher-level model to FlatZinc may be annotated thus:

```
var int: TMP :: var_is_introduced;
```

This information is potentially useful to the solver's search strategy.

4 Output

An implementation should output values for all and only the variables annotated with `output_var` or `output_array`.

Output should be in alphabetical order and take the following form:

varname = literal;

or, for array variables,

varname = `array`N`d(`x_1`..`x_2`, ..., [`y_1`, `y_2`, ...`y_k`]);`

where N is the number of index sets specified in the corresponding `output_array` annotation, x_1..x_2, ... are the index set ranges, and y_1, y_2, ... y_k are literals.

Implementations should ensure that *all* model variables (not just the output variables) have satisfying assignments before printing a solution.

The output for a solution must be terminated with ten consecutive minus signs on a separate line: `----------`.

Multiple solutions may be output, one after the other, as search proceeds.

If search terminates having explored the whole search tree, ten consecutive equals signs should be printed on a separate line: `==========` (output consisting only of this row of equals signs therefore indicates "no solution").

A Required FlatZinc Predicates

The type signature of each required predicate is preceded by its specification (n denotes the length of any array arguments).

$(\forall\ i \in 1..n :\ as[i]) \leftrightarrow r$
```
array_bool_and(array [int] of var bool: as, var bool: r)
```

$as[b] = c$
```
array_bool_element(array [int] of bool: as, var int: b, var bool: c)
```

$(\exists\ i \in 1..n :\ as[i]) \leftrightarrow r$
```
array_bool_or(array [int] of var bool: as, var bool: r)
```

$as[b] = c$
```
array_float_element(array [int] of float: as, var int: b, var float: c)
```

$as[b] = c$
```
array_int_element(array [int] of int: as, var int: b, var int: c)
```

$as[b] = c$
```
array_set_element(array [int] of set of int: as, var int: b, var set of int: c)
```

$as[b] = c$
```
array_var_bool_element(array [int] of var bool: as, var int: b, var bool: c)
```

$as[b] = c$
```
array_var_float_element(array [int] of var float: as, var int: b, var float: c)
```

$as[b] = c$
```
array_var_int_element(array [int] of var int: as, var int: b, var int: c)
```

$as[b] = c$
```
array_var_set_element(array [int] of var set of int: as, var int: b, var set of int: c)
```

$(a \leftrightarrow b = 1) \wedge (\neg a \leftrightarrow b = 0)$
```
bool2int(var bool: a, var int: b)
```

$(a \wedge b) \leftrightarrow r$
```
bool_and(var bool: a, var bool: b, var bool: r)
```

$(\exists\ i \in 1..n_{as} :\ as[i]) \vee (\exists\ i \in 1..n_{bs} :\ \neg bs[i])$
```
bool_clause(array [int] of var bool: as, array [int] of var bool: bs)
```

$((\exists\ i \in 1..n_{as} :\ as[i]) \vee (\exists\ i \in 1..n_{bs} :\ \neg bs[i])) \leftrightarrow r$
```
bool_clause_reif(array [int] of var bool: as, array [int] of var bool: bs, var bool: r)
```

$a = b$
```
bool_eq(var bool: a, var bool: b)
```

$(a = b) \leftrightarrow r$
```
bool_eq_reif(var bool: a, var bool: b, var bool: r)
```

$a \ \vee \ \neg b$
`bool_ge(var bool: a, var bool: b)`

$(a \ \vee \ \neg b) \ \leftrightarrow \ r$
`bool_ge_reif(var bool: a, var bool: b, var bool: r)`

$a \ \wedge \ \neg b$
`bool_gt(var bool: a, var bool: b)`

$(a \ \wedge \ \neg b) \ \leftrightarrow \ r$
`bool_gt_reif(var bool: a, var bool: b, var bool: r)`

$\neg a \ \vee \ b$
`bool_le(var bool: a, var bool: b)`

$(\neg a \ \vee \ b) \ \leftrightarrow \ r$
`bool_le_reif(var bool: a, var bool: b, var bool: r)`

$(a \ \leftarrow \ b) \ \leftrightarrow \ r$
`bool_left_imp(var bool: a, var bool: b, var bool: r)`

$\neg a \ \wedge \ b$
`bool_lt(var bool: a, var bool: b)`

$(\neg a \ \wedge \ b) \ \leftrightarrow \ r$
`bool_lt_reif(var bool: a, var bool: b, var bool: r)`

$a \ \neq \ b$
`bool_ne(var bool: a, var bool: b)`

$(a \ \neq \ b) \ \leftrightarrow \ r$
`bool_ne_reif(var bool: a, var bool: b, var bool: r)`

$\neg a \ = \ b$
`bool_not(var bool: a, var bool: b)`

$(a \ \vee \ b) \ \leftrightarrow \ r$
`bool_or(var bool: a, var bool: b, var bool: r)`

$(a \ \rightarrow \ b) \ \leftrightarrow \ r$
`bool_right_imp(var bool: a, var bool: b, var bool: r)`

$(a \ \neq \ b) \ \leftrightarrow \ r$
`bool_xor(var bool: a, var bool: b, var bool: r)`

$a \ = \ b$
`float_eq(var float: a, var float: b)`

$a \ \geq \ b$
`float_ge(var float: a, var float: b)`

$a \ > \ b$

`float_gt(var float: a, var float: b)`

$a \;\leq\; b$
`float_le(var float: a, var float: b)`

$\sum\; i \in 1..n:\;\; as[i].bs[i] \;\;=\;\; c$
`float_lin_eq(array [int] of float: as, array [int] of var float: bs, float: c)`

$\sum\; i \in 1..n:\;\; as[i].bs[i] \;\;\geq\;\; c$
`float_lin_ge(array [int] of float: as, array [int] of var float: bs, float: c)`

$\sum\; i \in 1..n:\;\; as[i].bs[i] \;\;>\;\; c$
`float_lin_gt(array [int] of float: as, array [int] of var float: bs, float: c)`

$\sum\; i \in 1..n:\;\; as[i].bs[i] \;\;\leq\;\; c$
`float_lin_le(array [int] of float: as, array [int] of var float: bs, float: c)`

$\sum\; i \in 1..n:\;\; as[i].bs[i] \;\;<\;\; c$
`float_lin_lt(array [int] of float: as, array [int] of var float: bs, float: c)`

$a \;<\; b$
`float_lt(var float: a, var float: b)`

$a - b \;=\; c$
`float_minus(var float: a, var float: b, var float: c)`

$-a \;=\; b$
`float_negate(var float: a, var float: b)`

$a + b \;=\; c$
`float_plus(var float: a, var float: b, var float: c)`

$|a| \;=\; b$
`int_abs(var int: a, var int: b)`

$\lfloor a/b \rfloor \;=\; c$
`int_div(var int: a, var int: b, var int: c)`

$a \;=\; b$
`int_eq(var int: a, var int: b)`

$(a \;=\; b) \;\leftrightarrow\; r$
`int_eq_reif(var int: a, var int: b, var bool: r)`

$a \;\geq\; b$
`int_ge(var int: a, var int: b)`

$(a \;\geq\; b) \;\leftrightarrow\; r$
`int_ge_reif(var int: a, var int: b, var bool: r)`

$a \;>\; b$
`int_gt(var int: a, var int: b)`

$(a \;>\; b) \;\leftrightarrow\; r$
`int_gt_reif(var int: a, var int: b, var bool: r)`

$a \;\leq\; b$
`int_le(var int: a, var int: b)`

$(a \;\leq\; b) \;\leftrightarrow\; r$
`int_le_reif(var int: a, var int: b, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;=\; c$
`int_lin_eq(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;=\; c) \;\leftrightarrow\; r$
`int_lin_eq_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;\geq\; c$
`int_lin_ge(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;\geq\; c) \;\leftrightarrow\; r$
`int_lin_ge_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;>\; c$
`int_lin_gt(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;>\; c) \;\leftrightarrow\; r$
`int_lin_gt_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;\leq\; c$
`int_lin_le(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;\leq\; c) \;\leftrightarrow\; r$
`int_lin_le_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;<\; c$
`int_lin_lt(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;<\; c) \;\leftrightarrow\; r$
`int_lin_lt_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$\sum\; i \in 1..n : \; as[i].bs[i] \;\neq\; c$
`int_lin_ne(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum\; i \in 1..n : \; as[i].bs[i] \;\neq\; c) \;\leftrightarrow\; r$
`int_lin_ne_reif(array [int] of int: as, array [int] of var int: bs, int: c, var bool: r)`

$a \;<\; b$
`int_lt(var int: a, var int: b)`

$(a \;<\; b) \;\leftrightarrow\; r$
`int_lt_reif(var int: a, var int: b, var bool: r)`

$\max(a,\; b) \;=\; c$
`int_max(var int: a, var int: b, var int: c)`

$\min(a,\ b)\ =\ c$
```
int_min(var int: a, var int: b, var int: c)
```

$a - b\ =\ c$
```
int_minus(var int: a, var int: b, var int: c)
```

$a - \lfloor a/b \rfloor.b\ =\ c$
```
int_mod(var int: a, var int: b, var int: c)
```

$a\ \neq\ b$
```
int_ne(var int: a, var int: b)
```

$(a\ \neq\ b)\ \leftrightarrow\ r$
```
int_ne_reif(var int: a, var int: b, var bool: r)
```

$-a\ =\ b$
```
int_negate(var int: a, var int: b)
```

$a + b\ =\ c$
```
int_plus(var int: a, var int: b, var int: c)
```

$ab\ =\ c$
```
int_times(var int: a, var int: b, var int: c)
```

$|a|\ =\ b$
```
set_card(var set of int: a, var int: b)
```

$a - b\ =\ c$
```
set_diff(var set of int: a, var set of int: b, var set of int: c)
```

$a\ =\ b$
```
set_eq(var set of int: a, var set of int: b)
```

$(a\ =\ b)\ \leftrightarrow\ r$
```
set_eq_reif(var set of int: a, var set of int: b, var bool: r)
```

$a\ \supseteq\ b\ \vee\ \min(a \bigtriangleup b) \in b$
```
set_ge(var set of int: a, var set of int: b)
```

$a\ \supset\ b\ \vee\ \min(a \bigtriangleup b) \in b$
```
set_gt(var set of int: a, var set of int: b)
```

$a \in b$
```
set_in(var int: a, var set of int: b)
```

$(a \in b)\ \leftrightarrow\ r$
```
set_in_reif(var int: a, var set of int: b, var bool: r)
```

$a \cap b\ =\ c$
```
set_intersect(var set of int: a, var set of int: b, var set of int: c)
```

$a\ \subseteq\ b\ \vee\ \min(a \bigtriangleup b) \in a$

```
set_le(var set of int: a, var set of int: b)
```

$a \subset b \lor \min(a \triangle b) \in a$

```
set_lt(var set of int: a, var set of int: b)
```

$a \neq b$

```
set_ne(var set of int: a, var set of int: b)
```

$(a \neq b) \leftrightarrow r$

```
set_ne_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \subseteq b$

```
set_subset(var set of int: a, var set of int: b)
```

$(a \subseteq b) \leftrightarrow r$

```
set_subset_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \supseteq b$

```
set_superset(var set of int: a, var set of int: b)
```

$(a \supseteq b) \leftrightarrow r$

```
set_superset_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \triangle b = c$

```
set_symdiff(var set of int: a, var set of int: b, var set of int: c)
```

$a \cup b = c$

```
set_union(var set of int: a, var set of int: b, var set of int: c)
```