MiniZinc: Towards A Standard CP Modelling Language

Nicholas Nethercote[1], Peter J. Stuckey[1], Ralph Becket[1], Sebastian Brand[1], Gregory J. Duck[1], and Guido Tack[2]

[1] National ICT Australia and the University of Melbourne, Victoria, Australia
{njn,pjs,rafe,sbrand,gjd}@csse.unimelb.edu.au
[2] Programming Systems Lab, Saarland University, Saarbrücken, Germany
tack@ps.uni-sb.de

Abstract. There is no standard modelling language for constraint programming (CP) problems. Most solvers have their own modelling language. This makes it difficult for modellers to experiment with different solvers for a problem.
In this paper we present MiniZinc, a simple but expressive CP modelling language which is suitable for modelling problems for a range of solvers and provides a reasonable compromise between many design possibilities. Equally importantly, we also propose a low-level solver-input language called FlatZinc, and a straightforward translation from MiniZinc to FlatZinc that preserves all solver-supported global constraints. This lets a solver writer support MiniZinc with a minimum of effort—they only need to provide a simple FlatZinc front-end to their solver, and then combine it with an existing MiniZinc-to-FlatZinc translator. Such a front-end may then serve as a stepping stone towards a full MiniZinc implementation that is more tailored to the particular solver.
A standard language for modelling CP problems will encourage experimentation with and comparisons between different solvers. Although MiniZinc is not perfect—no standard modelling language will be—we believe its simplicity, expressiveness, and ease of implementation make it a practical choice for a standard language.

1 Introduction

Many constraint satisfaction and optimisation problems can be solved by CP solvers that use finite domain (FD) and linear programming (LP) techniques. There are many different solving techniques, and so there are many solvers. Examples include Gecode [1], ECLiPSe [2], ILOG Solver [3], Minion [4], and Choco [5].

However, these solvers use different, incompatible modelling languages that express problems at varying levels of abstraction. This makes life difficult for modellers—if they wish to experiment with different solvers, they must learn new modelling languages and rewrite their models. A standard CP modelling language supported by multiple solvers would mitigate this problem.

A standard CP modelling language could also make solver benchmarking simpler. For example, if the CSPlib benchmark library problems [6] had their natural language specifications augmented with standard machine-readable descriptions, it could stimulate competition between solver writers and lead to improvements in solver technology.

For these reasons, we believe a standard CP modelling language is desirable. The main challenges in proposing such a language are (a) finding a reasonable middle ground when different solvers have such a wide range of capabilities—particularly different levels of support for global constraints—and (b) encouraging people to use the language. In this paper we make the following two contributions that we believe solve these two problems.

A CP modelling language suitable as a standard. Section 2 introduces MiniZinc, a medium-level declarative modelling language.[3] MiniZinc is high-level enough to express most CP problems easily and in a largely solver-independent way; for example, it supports sets, arrays, and user-defined predicates, some overloading, and some automatic coercions. However, MiniZinc is low-level enough that it can be mapped easily onto many solvers. For example, it is first-order, and it only supports decision variable types that are supported by most existing CP solvers: integers, floats, Booleans and sets of integers. Other MiniZinc features include: it allows separation of a model from its data; it provides a library containing declarative definitions of many global constraints; and it also has a system of annotations which allows non-declarative information (such as search strategies) and solver-specific information (such as variable representations) to be layered on top of declarative models.

A simple way to implement MiniZinc. Solver writers (who may not have language implementation skills) will not want to do a large amount of work to support a modelling language. Therefore we provide a way for solver writers to provide reasonable MiniZinc support with a minimum of effort. Section 3 introduces FlatZinc, a low-level solver input language that is the target language for MiniZinc. FlatZinc is designed to be easy to translate into the form required by a CP solver. Section 4 then defines a standard translation from MiniZinc to FlatZinc, which involves only well-understood transformations such as predicate inlining and reification. Importantly it allows a solver to use native definitions of any global constraints it supports, while decomposing unsupported ones into lower-level constraints. Even though this transformation will not be ideal for all solvers, it provides an excellent starting point for an implementation.

At the paper's end, Section 5 describes our supporting tool set and presents some experimental results, Section 6 presents related work, and Section 7 discusses future work and concludes.

The core MiniZinc language is not particularly novel, as is appropriate for a standard language—it incorporates ideas from many existing modelling languages. The novel features are: (a) the use of predicates to allow more extensible

[3] Modelling is sometimes divided into "conceptual" and "design" modelling. But these are just two points on a spectrum of "how many modelling decisions have been made" that spans from the highest level (e.g. natural language) to the lowest level (e.g. solver input formats). For this reason we follow the simpler programming language terminology and talk about high-, medium- and low-level languages.

modelling, (b) the use of annotations to provide non-declarative and solver-specific information, (c) the use of a lower-level language and a standard translation to make it easy to connect the language to existing solvers, and (d) the preservation of calls to supported global constraints in that translation.

We believe that MiniZinc's characteristics—simplicity, expressiveness, and ease of initial support—make it a practical choice for a standard language.

2 MiniZinc

2.1 Specifying a Problem

A MiniZinc problem specification has two parts: (a) the *model*, which describes the structure of a class of problems; and (b) the *data*, which specifies one particular problem within this class. The pairing of a model with a particular data set is a *model instance* (sometimes abbreviated to *instance*).

The model and data may be in separate files. Data files can only contain assignments to parameters declared in the model. A user specifies data files on the command line, rather than naming them in the model file, so that the model file is not tied to any particular data file.

2.2 A MiniZinc Example

Each MiniZinc model is a sequence of *items*, which may appear in any order. Consider the MiniZinc model and example data for a restricted job shop scheduling problem in Figures 1 and 2.

Line 0 is a comment, introduced by the '`%`' character.

Lines 1–5 are *variable declaration items*. Line 1 declares `size` to be an integer *parameter*, i.e. a variable that is fixed in the model. Line 20 (in the data file) is an *assignment item* that defines the value of `size` for this instance. Variable declaration items can include assignments, as in line 3. Line 4 declares `s` to be a 2D array of *decision variables*. Line 5 is an integer variable with a restricted range. Decision variables are distinguished by the `var` prefix.

Lines 7–8 show a user-defined *predicate item*, `no_overlap`, which constrains two tasks given by start time and duration so that they do not overlap in time.

Lines 10–17 show a *constraint item*. It uses the built-in `forall` to loop over each job, and ensure that: (line 12) the tasks are in order; (line 13) they finish before `end`; and (lines 14–16) that no two tasks in the same column overlap in time. Multiple constraint items are allowed, they are implicitly conjoined.

Line 19 shows a *solve item*. Every model must include exactly one solve item. Here we are interested in minimising the end time. We can also maximise a variable or just look for any solution ("`solve satisfy`").

There is one kind of MiniZinc item not shown by this example: *include items*. They facilitate the creation of multi-file models and the use of library files.

There is currently no way to control the output produced at run-time. We plan to add such control in the near future (see Section 7).

```
% (square) job shop scheduling in MiniZinc
int: size;                                  % size of problem
array [1..size,1..size] of int: d;          % task durations
int: total = sum(i,j in 1..size) (d[i,j]);  % total duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end;                          % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint
    forall(i in 1..size) (
        forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,size] + d[i,size] <= end /\
        forall(j,k in 1..size where j < k) (
            no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
        )
    );

solve minimize end;
```

Fig. 1. MiniZinc model (`jobshop.mzn`) for the job shop problem.

```
size = 2;
d = [ 2,5,
      3,4 ];
```

Fig. 2. MiniZinc data (`jobshop2x2.data`) for the job shop problem.

2.3 Types and Insts

MiniZinc provides three scalar types: Booleans, integers, and floats; and two compound types: sets, and arrays. There are no user-defined types, however we will see shortly that restricted types such as integer and float ranges are allowed. Scalars and sets have a built-in (lexicographical) ordering.

As well as having a type, each variable has an *instantiation* (often abbreviated to *inst*), which indicates if it is fixed in the model to a known value (a parameter, shortened to *par*) or not (a decision variable, shortened to *var*). A pairing of a type and an inst is called a *type-inst*.

Booleans, integers and floats may be parameters or decision variables. Example syntax for scalars: `par bool`, `var int`, `float`; if the inst is omitted it defaults to `par`. There is no automatic coercion of integers to floats.

Sets can only contain *par* scalars. Sets of integers can be *par* or *var*, but all other sets must be *par*. For example: `var set of int` is legal, but `var set of bool` and `set of var int` are illegal.

Arrays must be *par*, i.e. of fixed length. They can be multi-dimensional. Each dimension's index set is a contiguous range of integers. Arrays may contain *par* or *var* scalars or sets of integers. For example, `array[0..9,5..10] of var int` is a 2D array of integer decision variables.

The following set expressions can be used as types: set ranges, set literals, and *par* set variables. Float ranges can also be used as types. The meaning is as if the type was declared as a normal type and then constrained, for example:

```
0..3:         v1;      %     int: v1;  constraint v1 in 0..3;
var {1,3,5}: v2;       % var int: v2;  constraint v2 in {1,3,5};
var 0.1 .. 9.5: v4;    % var float: v4;
                       %   constraint 0.1 <= v4 /\ v4 <= 9.5;
```

MiniZinc has some polymorphism: some operations are overloaded to work with multiple type-insts (see Section 2.5); certain arrays are automatically coerced, and the type of each anonymous variable '`_`' (see Section 2.4) is inferred; and *par* values are automatically coerced to *var* values as necessary.

2.4 Expressions

MiniZinc has several kinds of expression.

Variable names can serve as expressions. Also, there is a special identifier '`_`' that represents an unconstrained, anonymous decision variable (of any type). It is particularly useful when partially initialising arrays, e.g. in a Sudoku puzzle.

Scalar literals are written in standard ways, for example: `true`, `false`, `23`, `-44.5`, `2.3e-05`.

Sets are written using set literals or set comprehensions. For example: `{1,2,3}` or `{ i * j  | i,j in 1..10 where i != j }`. Comprehensions can have multiple variables per generator, multiple generators, and each generator can have a filtering `where` clause.

Arrays are written similarly, using array literals or array comprehensions. For example: `[2,_,3,_]` or `[ 2*i | i in 1..5 ]`. Array literals are automatically coerced to different index ranges so long as the element type and lengths match; lines 21 and 22 of the job shop model shows an example with a 2D array.

If-then-else expressions (e.g. `if C1 then A else B endif`) and let expressions (e.g. `let { int: x = 1, int: y = 2 } in x + y`) are supported; the latter are used within predicates to declare local variables (see Section 2.6). The condition of an if-then-else must be *par*.

Most predicate and function calls use the usual syntax, e.g. `even(n)`. Some built-in functions, such as '`+`', are *operators*—their names are non-alphanumeric, and calls to them are written using infix or prefix notation. There is also special syntax for combining an array comprehension with a call—a *generator call* `P(Gs)(E)` is equivalent to `P([E | Gs])`; the parentheses around the `E` are mandatory so as to avoid possible ambiguity when the generator call is part of a larger expression. Figure 1 includes an example, `sum(i,j in 1..size)(d[i,j])`, which is syntactic sugar for `sum([d[i,j] | i,j in 1..size])`.

2.5 Built-in Operations

MiniZinc has many useful built-in operators, predicates and functions. They include: comparisons (e.g. `<`, `==`), arithmetic operations (e.g. `+`, `*`, `sum`, `min`), logical operations (e.g. `/\`, `xor`, `forall`), set operations (e.g. `union`, `subset`, `in`, `card`), array operations (e.g. `length`, `index_set`), coercions (e.g. `round`, `int2float`, `bool2int`), and bounds operations (`ub`, `lb`, `dom`).

Most are overloaded to work with parameters and decision variables. Some are overloaded to work with multiple types, e.g. arithmetic operations work with both integers and floats, and comparisons support all types.

2.6 Predicates

Users can define their own predicates in MiniZinc. The job shop model shows an example, `no_overlap`. Predicates may not be recursive, but can call other predicates. Predicates implicitly return a Boolean value. Predicates may have local variables; they are introduced via let expressions. For example:

```
predicate even(var int:x) = let { var int: y } in x == 2 * y;
```

Any predicate containing a non-*par* local variable cannot be called in a possibly-negated context (e.g. inside a `not`, or `<->`), because the local variable(s) would be effectively universally quantified, which most solvers do not support.

2.7 Global Constraints

MiniZinc has a global constraints library. It can be used in models via an include item: `include "globals.mzn"`. The default version of this library contains predicate definitions of many global constraints, such as `all_different`, `cumulative`, etc.

Crucially, this library can be tailored by solver writers for use with individual solvers. All global constraints that are not supported by the solver should be left untouched, so calls to them can be inlined during the MiniZinc-to-FlatZinc translation. For example:

```
predicate disjoint(var set of int:S, var set of int:T) =
    S intersect T == {};
```

In contrast, global constraints that are supported by a solver can have their definition removed (but not their declaration). Calls to these global constraints will be left untouched by the MiniZinc-to-FlatZinc conversion.

Type-overloaded global constraints that are supported by the solver can be defined separately for each type. For example, the overloaded `all_different` might be defined as:

```
predicate all_different(array[int] of var float:x) =
    forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
predicate all_different(array[int] of var int:x) =
    gecode_all_different(x);  % native Gecode version for ints
```

```
30  array [1..size,1..size] of var 0..total: a :: bounds;
31  constraint all_different(a) :: domain :: priority(length(a));
32  solve :: int_search([end],"input_order","indomain_min","complete")
33        :: int_search(a,"smallest","indomain_min","lds(3)")
34      minimize end;
```

Fig. 3. Example annotations on variables, constraints and solve items.

Finally, although MiniZinc provides a standard definition for each global constraint, a solver writer can arbitrarily replace each definition with an alternative definition in MiniZinc that may suit their solver better, if that is easier than providing a native implementation.

2.8 Modelling Techniques

MiniZinc is deliberately small to make it easier to translate. Some common CP modelling techniques need to be indirectly modelled in MiniZinc.

Enumerated types can be modelled using named sets of integers, for example:

```
set of int: Colour = 1..3;
Colour: Red = 1;  Colour: Green = 2;  Colour: Blue = 3;
array[Colour,Colour] of var int: clashing;
```

Extensional relations can be modelled using multiple arrays with the same indices, e.g. $(x, y) \in \{(1, 2.0), (-2, 3.4), (6, -1000.0)\}$ is modelled as:

```
array[1..3] of int:   r1 = [1,  -2,    6];
array[1..3] of float: r2 = [2.0, 3.4, -1000.0];
var 1..3: i;  var int: x;  var int: y;
constraint x == r1[i] /\ y == r2[i];
```

However this does not allow negatively defined relations such as $(x, y) \notin \{(5, 6), (8, 9)\}$.

2.9 Adding Non-declarative and Solver-specific Information

MiniZinc (as described so far) includes no information about how models should be solved. In practice we need a way to attach such non-declarative and solver-specific information to the model to support efficient solving.

Our solution is to use *annotations*. MiniZinc defines some standard annotations that should be supported by most solvers. Also, because a solver is free to ignore annotation information, more solver-specific annotations can be used.

Annotations consist of an identifier, with optional arguments appearing in parentheses. We use string literals for many annotation arguments. Annotations are attached to variable declarations, expressions, and solve items using the (left-associative) operator `::` which binds tighter than all other operators, as illustrated by the examples in Figure 3.

Line 30 shows a variable annotations. Here we assume that `bounds` instructs the solver to use a special implementation that only maintains bounds.

Line 31 shows two constraint annotations: the first (`domain`) instructs the solver to use a domain-consistent (GAC) version of the `all_different` constraint; the second (`priority`) assigns the constraint a priority equal to the size of its argument array. Another constraint annotation is `bounds`, which indicates that bounds propagation should be used for the constraint.

Lines 32–34 give an example of an annotated solve item, where the search strategy is specified for an FD solver. Solve annotations are based on the `search` predicate of ECLiPSe. The parameters to the `int_search` annotation indicate: (a) the variables being fixed via the strategy, (b) the variable selection strategy, (c) the value choice method, and (d) the exploration strategy. Combinations of strategies can be specified in order. The strategy in the example is: first set `end` to its least value and then try setting start times by setting the variable with the smallest possible value to this value, and only consider limited discrepancy search with a limit of 3 discrepancies.

3 FlatZinc

FlatZinc is mostly a subset of MiniZinc. We wait until Section 4 before giving a FlatZinc example, in order to show how MiniZinc-to-FlatZinc translation works.

Model structure. Unlike MiniZinc, FlatZinc has no model/data separation, nor multi-file models—a FlatZinc model instance must be in a single file.

Items. Some of the MiniZinc items are supported: constraint items, variable declarations (with optional assignments) and solve items. The rest are not: include items, stand-alone assignment items, and user-defined predicates. Unlike MiniZinc, in FlatZinc variables must be defined before they are used.

Types and type-insts. Some MiniZinc types (and their insts) are supported: Booleans, integers and floats (including range-restricted ones), and sets. Arrays are supported but must be one-dimensional, and they are always indexed from $0..length - 1$. Also, there is no type or type-inst polymorphism—for example, the built-in `int_plus` is distinct from `float_plus`. Implicit *par*-to-*var* coercions are supported, as in MiniZinc.

Expressions. Some of MiniZinc's expressions are supported: identifiers, scalar literals, set literals, array literals, predicate calls, and array accesses with a *par* index. Expressions not supported are: anonymous variables, set and array comprehensions, if-then-else expressions, let expressions, generator calls, and array accesses with a *var* index (which must be done via `element` constraints). Also, no operators are supported (but negative numbers such as `-1` are allowed—the '`-`' is considered part of the numeric literal.)

Built-ins. The main way in which FlatZinc is not a subset of MiniZinc is that it has different built-in operations. These are operations that a CP solver is expected to support natively. Most of them correspond directly to a MiniZinc operation, although the names are different because FlatZinc has no operators or overloading. They include: comparison constraints (e.g. `int_eq`, `float_gt`), linear (in)equalities (e.g. `int_lin_eq`), arithmetic constraints (e.g. `int_plus`),

```
40  array[0..3] of var 0..14: s;
41  var 0..14: end;
42  var bool: b1;
43  var bool: b2;
44  var bool: b3;
45  var bool: b4;
46  constraint  int_lin_le     ([1,-1], [s[0], s[1]], -2);
47  constraint  int_lin_le     ([1,-1], [s[2], s[3]], -3);
48  constraint  int_lin_le     ([1,-1], [s[1], end ], -5);
49  constraint  int_lin_le     ([1,-1], [s[3], end ], -4);
50  constraint  int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1);
51  constraint  int_lin_le_reif([1,-1], [s[2], s[0]], -3, b2);
52  constraint  bool_or(b1, b2, true);
53  constraint  int_lin_le_reif([1,-1], [s[1], s[3]], -5, b3);
54  constraint  int_lin_le_reif([1,-1], [s[3], s[1]], -4, b4);
55  constraint  bool_or(b3, b4, true);
56  solve minimize end;
```

Fig. 4. FlatZinc translation of the MiniZinc job shop model.

logical constraints (e.g. `bool_or`, `bool_not`), set constraints (e.g. `set_subset`, `set_card`), element constraints (e.g. `array_int_element`), and coercion constraints (e.g. `bool2int`). There are also reified versions of many constraints which take an additional Boolean argument, e.g. `int_eq_reif`, `set_subset_reif`.

Also, a FlatZinc model instance may include calls to any global constraints that the target solver supports natively, as Section 2.7 explained.

Annotations. FlatZinc's annotations are the same as MiniZinc's, although any expressions within them must of course be valid FlatZinc expressions.

Writing a FlatZinc front-end. A FlatZinc front-end for a solver must parse the FlatZinc, and translate declarations and constraints into whatever form the solver requires. The grammar can be expressed in a way that most type and inst errors manifest as syntax errors, which reduces the work that must be done by the FlatZinc front-ends. Any FlatZinc constraints not handled by the solver can be converted into run-time aborts. These steps are easy by language implementation standards, because FlatZinc is so simple. Section 5 describes how our existing tools help further with this task. A solver writer must also specialise `globals.mzn`, which is a trivial exercise in removing predicate bodies.

4 Translating MiniZinc to FlatZinc

The translation from MiniZinc to FlatZinc has two parts: flattening, and the rest. We use the FlatZinc translation in Figure 4 of the MiniZinc model instance from Section 2.2 as an example. Line 40 is the original 2D array of decision variables, mapped to a zero-indexed 1D array. Line 41 is the original `end` variable. Lines 42–45 are variables introduced by Boolean decomposition. Lines 46–55 are the constraints. Lines 46 and 47 result from line 12, lines 48 and 49 result from line 13, and lines 50–55 result from lines 14–15 and 7–8.

4.1 Flattening

Flattening involves the following simple steps that statically reduce the model and data as much as possible. There is no fixed order to the steps because some enable others, which can then enable further application of previously applied steps. Therefore, they must be repeated, e.g. by iterating until a fixpoint is reached, or by re-flattening child nodes of expressions that have been flattened.

Parameter substitution. This step substitutes any atomic literal value assigned to a global or let-local scalar parameter throughout the model, and removes the declaration and assignment. For example, with `size = 2` we substitute `2` for `size`, but `size = 2 + y` would not be substituted until fully reduced.

Built-ins evaluation. This step evaluates all calls to built-ins that have fixed, atomic literal arguments. For example, `2-1` (from `size-1`, after parameter substitution) in the jobshops example becomes `1`.

Comprehension unrolling. This step unrolls all set and array comprehensions, once the generator ranges are fully reduced.

Compound built-in unrolling. This step unrolls compound built-ins (those that involve the folding of an operation over an array of elements, such as `sum` and `forall`) by replacing them with multiple lower-level operations.

We will use lines 11, 14 and 15 of Figure 1 as the starting point of a running example. They unroll to give the following conjunction (the first conjunct has `i=1`, `j=1` and `k=2`; the second has `i=2`, `j=1` and `k=2`).

```
no_overlap(s[1,1], d[1,1], s[2,1], d[2,1]) /\
no_overlap(s[1,2], d[1,2], s[2,2], d[2,2])
```

Fixed array access replacement. This step replaces all array accesses involving fixed indices and fixed elements with the appropriate value. Once all the accesses of an array have been replaced, its declaration and assignment can be removed. For example, our running example becomes:

```
no_overlap(s[1,1], 2, s[2,1], 3) /\
no_overlap(s[1,2], 5, s[2,2], 4)
```

If-then-else evaluation. This step evaluates each if-then-else expression, once its condition is fully reduced. This is always possible because if-then-else conditions must be fixed.

Predicate inlining. This step replaces each call to a defined predicate with its body, substituting actual arguments for formal arguments. This is easy because predicates cannot be recursive, either directly or mutually. Calls to predicates lacking a definition (such as those in the MiniZinc globals library) are left as-is. For example, the first conjunct from our running example becomes:

```
s[1,1] + 2 <= s[2,1] \/ s[2,1] + 3 <= s[1,1]
```

4.2 Post-flattening

After flattening, we apply the following steps once each, in the given order.

Stand-alone assignment removal. This step removes each stand-alone assignment by merging it with the appropriate variable declaration.

Let floating. This step moves let-local decision variables to the top-level and renames them appropriately.

Boolean decomposition. This step decomposes all Boolean expressions that are not top-level conjunctions. It replaces each sub-expression with a new Boolean variable (also adding a declaration for each variable), and adds conjuncts equating these new variables with the sub-expressions they replaced. This facilitates the later introduction of reified constraints. For example, our running example becomes:

```
((b1 \/ b2) <-> true) /\
((s[1,1] + 2 <= s[2,1]) <-> b1) /\
((s[2,1] + 3 <= s[1,1]) <-> b2)
```

Declarations are also added for the new Boolean variables `b1` and `b2`.

Numeric decomposition. This step decomposes numeric equations or inequations in a manner similar to Boolean decomposition, by renaming each non-linear sub-expression with a new variable.

Set decomposition. This step decomposes compound set expressions into primitive set constraints in a manner similar to Boolean/numeric decomposition.

(In)equality normalisation. This step normalises (in)equations, e.g. it converts `>=` into `<=`, moves sub-expressions so the right-hand side is constant, and replaces negations with multiplications by -1. This facilitates the later introduction of linear (in)equality constraints. For example, the second conjunct from our running example becomes:

```
(s[1,1] + (-1)*s[2,1] <= -2) <-> b1
```

Array simplification. This step simplifies all arrays to one-dimensional, zero-indexed arrays. It also updates any remaining array accesses accordingly. For example, our running example becomes:

```
(s[0] + (-1)*s[2] <= -2) <-> b1
```

Anonymous variable naming. This step replaces each anonymous variable (‘`_`’) with a newly introduced variable of the appropriate type.

Conversion to FlatZinc built-ins. This step converts the remaining MiniZinc built-ins and array accesses (which all must have at least one non-*par* argument) into FlatZinc built-ins. The FlatZinc built-ins may be type-specialised, and will be reified if the MiniZinc built-in occurs inside a Boolean expression (other than conjunction).

The most complex case involves linear (in)equalities. Each one is replaced with a (type-specific, possibly reified) linear predicate, unless it can be replaced with a simpler arithmetic constraint. For example, our running example becomes:

```
int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1)
```

The next case is that array accesses involving non-fixed indices are replaced with FlatZinc element constraints such as `array_int_element`.

The remaining cases are simpler, as each remaining MiniZinc operation has a corresponding FlatZinc operation. For example, `(b1 \/ b2) <-> true` becomes `bool_or(b1, b2, true)`, and `x * y = z` becomes `int_times(x, y, z)`.

Top-level conjunction splitting. This step splits top-level conjunctions into multiple constraint items, e.g. `constraint a /\ b;` becomes two items.

4.3 Annotations

Most annotations are maintained during the translation. Fixed expressions within annotations are evaluated like other expressions. When annotated expressions are unrolled, the annotation is copied to the resulting operations. When expressions are reified, their annotations are lost.

4.4 Summary

The translation above provides much of what a solver writer would want. But it is clearly not ideal for every underlying solver. For example, solvers may be more efficient on the undecomposed versions of Boolean constraints [7] or non-linear constraints. We plan to investigate methods to control which transformations should be applied for a particular solver when we have more experience.

5 Tool Set and Experiments

We have a MiniZinc front-end that parses, checks types and instantiations, and converts to FlatZinc. It has two different MiniZinc-to-FlatZinc converters: one written using the term-rewriting system Cadmium [8] (which produced Figure 4 as shown, modulo some variable renaming and item reordering), and the other written in Mercury. The Cadmium implementation allows us to omit various steps of the translation.

Our MiniZinc global constraints library includes around a dozen global constraints, all less than 20 lines of code. Clearly there are many more to be included.

We have several FlatZinc front-ends. The first one, FlatZinc/G12, is a FlatZinc interpreter for the G12 constraint platform. It uses FD and FD set solvers written in Mercury, and one of several LP solvers such as CPLEX and GLPK. The second front-end, FlatZinc/Gecode, is a FlatZinc interpreter for Gecode [1]. It was implemented from scratch in less than one week by one of the authors (a Gecode developer) with no prior knowledge of FlatZinc. Reuse of the lex and yacc parser he developed should further reduce development time for other solver writers. The third front-end, FlatZinc/Eclipse, is a prototype front-end for ECLiPSe that plugs into FlatZinc/G12; it was written in one afternoon using Cadmium. Finally, we also have a translator from FlatZinc to the Minion

	Size (LOC)			Trans. time (s)		Solve time (s)			
Benchmark	MZ	FZ	Ge	Merc	Cd	Ge	FZ/Ge	FZ/G12	FZ/Ecl
alpha	54	55	65	0.05	0.70	0.22	0.23	0.35	0.67
eq20	66	82	61	0.12	0.68	0.00	0.00	0.01	0.02
packing2	41	1145	138	0.12	0.73	0.02	0.14	0.15	0.52
warehouses	47	517	100	0.14	0.98	0.00	0.02	0.79	0.04

Fig. 5. Experimental Results. Column 1 gives the benchmark names. Columns 2–4 give the code sizes of the MiniZinc, FlatZinc and native-Gecode versions. Column 5–6 give the translation times for the Mercury and Cadmium MiniZinc-to-FlatZinc translators. Columns 7–10 give the solve times for the native Gecode, FlatZinc/Gecode, FlatZinc/G12 and FlatZinc/Eclipse versions.

format [4], but we do not present results for it. Although Minion's file format appears similar to FlatZinc, the conversion is non-trivial: Minion offers various tuning options (notably a choice of operationally different variable types and constraint propagators), and it allows only basic control over search.

Figure 5 shows some results comparing native Gecode (v1.3.1) to FlatZinc/Gecode and FlatZinc/G12. The benchmarks were taken from the Gecode examples suite and ported to MiniZinc. All implementations use the same search strategy for each benchmark. The test machine was a 2.0 GHz Pentium M with 2GB of RAM and 2MB L2 cache running Fedora Core 4 (Linux kernel 2.6.15). All Mercury code was compiled with Mercury rotd-2007-02-05, and C and C++ code was compiled with GCC 4.0.2. All timings are the best of five runs.

The code sizes show that MiniZinc models are compact, much more so than native Gecode programs—Gecode is not a modelling language, its constraints are written as low-level C++ calls to the Gecode library, and a Gecode model is thus a C++ program. The code sizes also show that FlatZinc models are (unsurprisingly) bigger than MiniZinc models, sometimes greatly so; this is due to FlatZinc's lack of looping constructs.

The translation times show that the MiniZinc-to-FlatZinc translation is fast in Mercury, but slower in Cadmium. Note that the translation step itself only accounts for part of the time taken—these numbers include the overhead of parsing, topological sorting and type-checking of the MiniZinc code, as well as printing of the FlatZinc. Also, these programs have not been tuned and so there is scope for further speed improvement.

The FlatZinc/Gecode solve time results show that FlatZinc models are competitive with the native Gecode models, and thus that the MiniZinc-to-FlatZinc translation is reasonable.

Although there is clearly more work to be done with other benchmarks and other front-ends, these results provide some evidence that MiniZinc and FlatZinc provide a way to write reasonably efficient models. It was difficult to ensure that the native Gecode versions were equivalent to the MiniZinc versions, because Gecode's search specifications are subtly different to MiniZinc's. This is more evidence that having a standard language would help with benchmarking.

The Gecode front-end is available at `www.gecode.org/flatzinc.html`. The other tools, the MiniZinc benchmarks, and a full specification of MiniZinc and FlatZinc are available at `www.g12.csse.unimelb.edu.au/minizinc/`.

6 Related Work

MiniZinc is (mostly) a subset of Zinc [9], a solver-independent modelling language designed to allow very high-level modelling and user-controlled translation of each high-level model to an appropriate solver-level model. Although we believe Zinc would be an excellent standard CP modelling language, the language is large enough that implementing it is a serious challenge. MiniZinc removes much of the complexity of Zinc—particularly user-defined types, various coercions, and user-defined functions—in order to make compilation simpler. Essence [10] and ESRA [11] are two other high-level, solver-independent modelling languages. MiniZinc should provide a good target language for them (as well as for Zinc).

MiniZinc is closely related to OPL [12]. Indeed, a cut-down version of OPL was proposed some time ago as a basis for a standard CP modelling language [13]. Compared to OPL, MiniZinc lacks complex types, resources, and programmable search specifications, among other things. The advantages of MiniZinc over OPL are its simplicity, the independence of models from global constraints supported by the solver, and the ease with which solver writers can utilise it (thanks to FlatZinc). Thus it should be easier to support as a proposed standard.

FlatZinc is similar to several low-level solver formats. The most significant of these is the verbose, XML format used by the CSP solver competitions [14]. Important differences with MiniZinc are: it has no separation of model and data, it is restricted to integers, it lacks arrays and looping constructs, and it has no solution to the problem of varied global constraints. Compared to this format, MiniZinc is more expressive and concise, and FlatZinc allows a similarly easy implementation for solver writers.

7 Conclusion and Future Work

Our main goal with MiniZinc was to define a language that is not too big but expressive enough to succinctly capture most CP problems. Our hope is that by providing simple-to-use tools for manipulating MiniZinc and FlatZinc it will be easy for solver writers to support this proposed standard, which will lead to benefits for both solver users and solver writers.

There are some obvious ways to extend MiniZinc which could be contemplated. Some of these are features of Zinc, for example: more type and instance polymorphism, user-defined functions, tuple types, and output items that allow control of the output from the solver. All involve a trade-off between modelling ease and implementation complexity. Other possible extensions include: a module system; a way of selectively applying parts of the MiniZinc-to-FlatZinc translation (e.g. omitting Boolean decomposition for solvers that handle Boolean constraints more directly); and the application of common sub-expression elimination to the MiniZinc-to-FlatZinc translation. Experience will guide the introduction of such extensions.

Finally, we intend to write FlatZinc translators to SAT and MIP solver input languages to further ease comparisons between different solving technologies.

Acknowledgments

Thanks to Christian Schulte, Mikael Lagerkvist, Joachim Schimpf, Kim Marriott and Mark Wallace for helpful discussions about MiniZinc and FlatZinc, and the anonymous reviewers for feedback on earlier versions of this paper.

References

1. Schulte, C., Lagerkvist, M., Tack, G.: Gecode. http://www.gecode.org/
2. Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2006)
3. ILOG: ILOG Solver. http://www.ilog.com/products/solver/
4. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast, scalable constraint solver. In: Proceedings of ECAI 2006, Riva del Garda, Italy (August 2006)
5. Laburthe, F.: CHOCO: implementing a CP kernel. In: Proceedings of TRICS 2000, Singapore (2000) 71–85
6. Gent, I.P., Walsh, T.: CSPLIB: a benchmark library for constraints. In: Proceedings of CP'99, Alexandria, Virginia, USA (October 1999) 480–481
7. Brand, S., Yap, R.H.: Towards "propagation = logic + control". In: Proceedings of ICLP 2006. Number 4079 in LNCS, Springer-Verlag (August 2006) 102–116
8. Duck, G., Stuckey, P., Brand, S.: ACD term rewriting. In: Proceedings of ICLP 2006. Number 4079 in LNCS, Springer-Verlag (August 2006) 117–131
9. Garcia de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: Proceedings of CP 2006, Nantes, France (September 2006) 700–705
10. Frisch, A.M., Grum, M., Jefferson, C., Hernandez, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: Proceedings of IJCAI-07, Hyderabad, India (January 2007)
11. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Proceedings of LOPSTR 2003, Uppsala, Sweden (August 2003) 214–232
12. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press (1999)
13. Wallace, M.: Personal communication (January 2007)
14. van Dongen, M., et al.: Second international CSP solver competition. http://cpai.ucc.ie/06/Competition.html