MiniZinc Discussion Paper

Peter J. Stuckey and the G12 team

NICTA Victoria Laboratory
University of Melbourne, 3010 Australia

Abstract. MiniZinc arose as a response to the extended discussion at CP2006 of the need for a standard modelling language for CP. This is a challenging problem, and we believe MiniZinc makes a good attempt to handle the most obvious obstacle: there are hundreds of potential global constraints, most handled by few or no systems. MiniZinc has come some way from its origins in 2007, and there are plenty of ways forward. This paper sets out the current state of MiniZinc and points out possible future directions.

1 Current Status

MiniZinc [1, 2] was our response to the call for a standard CP modelling language. 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.

Crucially for a standard to become realised it must be as simple as possible for solver writers to support the standard. This is the fundamental design goal of MiniZinc. MiniZinc models are translated to 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. We provide many features in order to make this translation specializable for a particular solver: including a modifiable library of global constraint definitions, facilities to rewrite the base FlatZinc constraints, and annotations to transport information from the model to the solver.

Currently MiniZinc is supported by a number of solvers developed by the G12 research team: `G12fd`, `lazyfd`, and `Chuffed`; as well as interfaces to external solvers developed by the G12 research team: `CPLEX` [3], `Gurobi` [4], `OSI-CBC` [5] and `MiniSAT` [6]; as well as interfaces to external solvers developed by others: `Gecode` [7], `ECLiPSe` [8], `SICStus Prolog` [9], `JaCoP` [10], `fzntini` [11], `SCIP` [12],

`fzn2smt` [13], `B-Prolog` [14], and `BEE` [15]. Encouragingly this includes CP systems like `Gecode` and `JaCoP`, CLP systems like `SICStus Prolog` and `B-Prolog`, the MIP-based system `SCIP`, translators to SAT like `fzntini` and `BEE`, and a translator to SMT, `fzn2smt`.

We have run the MiniZinc Challenge every year since 2008, to compare different solvers on the same benchmarks, one of the primary advantages of a common modelling language. An ancillary advantage of the Challenge is it forces us to collect and develop new MiniZinc benchmarks. There are now a wide range of benchmarks included in the MiniZinc release, and more are available through the MiniZinc Wiki [16], or from various websites (e.g. [17, 18]).

The MiniZinc release comes with a comprehensive test suite to test out solvers that support FlatZinc, and tools to run the test suite easily. We believe this makes it easy for solver writers to develop a FlatZinc interface, and should also provide new solver writers the opportunity to test their solver over a wide range of benchmarks easily.

The MiniZinc tool set is completely open source, although since it is written in Mercury [19] so far the only contributions to the source are from the G12 group. At present there is no public source repository for the tool set.

Significant extensions of the MiniZinc toolset since the original publication are:

- Modifying the translation from MiniZinc to FlatZinc so that it respects the *relational semantics* [20]
- Allowing the solver specific rewriting of FlatZinc builtins (analogous to how globals are rewritten)
- The addition of annotation declarations and annotation variables types.
- The addition of annotations: `defines_var` and `defines` during flattening to allow the original structure of flattened constraints to be recaptured
- Deprecation of many FlatZinc builtins which could be rewritten by a single equivalent FlatZinc constraint.

The remainder of this paper is organized as follows. In the next section we discuss very concrete possible extensions to MiniZinc that have arisen as we have gained much more experience in modelling and solving with the language. In Section 3 we discuss broader language design issues. In Section 4 we discuss how to improve the MiniZinc tool set. In Section 5 we discuss issues with organizing and proselytizing MiniZinc. Finally in Section 6 we conclude.

2 MiniZinc Concrete Issues

The first and most obvious topic for discussion for the future of MiniZinc are the possible concrete improvements that should be considered. Many of these are not difficult to add, they simply require the effort to be made by someone; others hide challenging underlying questions about meaning of models or whether an effective implementation is possible.

2.1 More Global Constraints

There are over 350 global constraints in the global constraint catalog (GCC) [21], but many of them are hardly used. Presently the MiniZinc globals include the most used 6: `alldifferent`, `cumulative`, `element`, `global_cardinality_low_up`, `regular`, and `table`; as well as a number of others. There is no reason not to add further globals to the language, we simply need to define a (hopefully relatively efficient) decomposition.

The only issue in adding new globals are:

1. naming—we should follow the name and argument order in the GCC;
2. extended types—types such as tuples are not supported by MiniZinc so we need to re-express the e.g. arrays of pairs as a pair of arrays; and
3. meta-level constraints—meta level constraints such as `geost` are not expressible in MiniZinc so they cannot be added at present.

Obvious candidates are: `circuit`, `cycle`, `disjoint`, `element_sparse`, `path`, and special case 0 constraints like `alldifferent_except_0` and `minimum_except_0`. Of course there is little point in adding new global constraints to the MiniZinc globals if no solver supports them natively.

Proposal 1. Every global supported by a solver that already supports MiniZinc should be added to the library. The task should hopefully be undertaken by the authors of the solver that supports the global. This would make the MiniZinc global library far more complete, and also spur the extension of solvers since these globals could then appear in the MiniZinc Challenge!

2.2 Extra Builtin Functions

Now that we have a great deal more experience with modelling with MiniZinc, and are aware of common idiom, there are clearly some missing functions that would be used frequently if they were there:

Min/Max

```
function var int: min(array[int] of var int: xs);
function var int: max(array[int] of var int: xs);
```

and likewise for parameters and `(var) float` are clearly just missing and should be added.

```
function var int: index_min(array[int] of var int: xs);
function var int: index_max(array[int] of var int: xs);
```

and likewise for parameters and `(var) float` are possible. These return the index of where the min or max exists. This is useful for handling pairs of arrays that model arrays of pairs, e.g.

```
array[1..10] of var 0..10: size;
array[1..10] of var 0..10: posn;
var int: i = index_min(size);
constraint posn[i] = 1;
```

FlatZinc predicates
`predicate array_int_min(array[int] of var int: xs, var int: m);`
`predicate array_int_max(array[int] of var int: xs, var int: m);`
are required to efficiently support the min and max functions, and can be used to replace the globals `minimum` and `maximum`.

Count
`function var int: count(array[int] of var bool: bs);`
which sums up an array of Boolean variables would be highly used and could translate to a new pseudo-Boolean FlatZinc constraint.

Xorall/Iffall
`function var bool: xorall(array[int] of var bool: bs);`
`function var bool: iffall(array[int] of var bool: bs);`
are natural n-ary Boolean functions which again can translate to a new Boolean FlatZinc constraint
`predicate array_bool_xor(array[int] of var bool: bs);`
This constraint can be implemented very efficiently (for example as in Crypto-MiniSat).

Output
`function string: concat(array[int] of string: xs);`
would make writing output items easier by allowing strings to be built using array comprehensions and then concatenated. An alternative is to introduce an array flattening function.

`function string: show_int(int: w, var int: x);`
which outputs an integer right justified in w characters, or left justified in $-w$ characters, and
`function string: show_float(int: w, int: m, var float: x);`
which outputs a float as $\%fw.m$ would make it easier to write complex output.

Proposal 2. Add all the above to MiniZinc except `index_min/max` which are probably not nearly as useful.

2.3 Search

Clearly the MiniZinc search language is far too limited. But adding stronger search features (particularly truly programmable search) adds a very serious burden to the solver writer.

For complex search strategies the *solver combinator* approach defined elsewhere in the workshop seems like a sensible approach to follow.

A more mundane task is to extend the base search facilities of MiniZinc. Some concrete suggestions on this line are:

– Remove the redundant fourth argument (which is always `complete`) from `int_search`, `bool_search` etc.
– Add variable choice annotations: `default` let the solver choose, `domwdeg`, `impact` and `activity`
– Add domain splitting annotations: `default`, `impact` and `activity`.

Finally we should consider search control annotations for MIP and SAT solvers (although these tend to rely on default search).

2.4 Half Reification

Half-reification (see the paper [22] in the main conference) can be advantageous for a number of reasons. A small example is

```
constraint x != y  \/ y != z;
```

which with full reification becomes

```
constraint int_neq_reif(x,y,b1);
constraint int_neq_reif(y,z,b2);
constraint bool_or(b1,b2,true);
```

but with half reification is

```
constraint int_neq_half(x,y,b1);
constraint int_neq_half(y,z,b2);
constraint bool_or(b1,b2,true);
```

The difference is that the fully reified constraints, e.g. $b1 \Leftrightarrow x \neq y$, wake whenever the domains of x, y and z change while the half-reified ones, e.g. $b1 \Rightarrow x \neq y$, only wake when one of the variables is fixed.

We plan to support it in an upcoming release of `mzn2fzn` controlled by a runtime flag, but it means extending solvers to handle "half-reified" versions of constraints $b \rightarrow c$. All the `reif` predicates (plus some Boolean predicates like `bool_xor`) will get an additional half-reified version

Half-reified constraints like
`predicate set_in_half(var int: x, var set of int: S, var bool: b);`
which encodes $b \rightarrow x \in S$ should be added for each constraint (perhaps with a different suffix, such as `_halfreif`).

By default these can be managed by FlatZinc decompositions, e.g.

```
predicate set_in_half(var int: x, var set of int: S, var bool: b) =
          b -> x in S;
```

A more complex question is the treatment of globals. Ideally we can persuade solver writers to define half-reified version of globals propagators since they are no more complex to build than unreified versions. Indeed we can then use the half-reified version to implement the unreified ones.

```
predicate alldifferent_half(array[int] of var int: x, var bool: b);
predicate alldifferent(array[int] of var int: x) =
          alldifferent_half(x, true);
```

By default we could use decompositions for the half-reified versions:

```
predicate alldifferent_decomp(array[int] of var int: x) =
          forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
predicate alldifferent_half(array[int] of var int: x, var bool: b) =
          b -> alldifferent_decomp(x);
```

2.5 Pseudo-Boolean Constraints

Pseudo-Boolean constraints occur very frequently in MiniZinc models, for example in constraints such as

```
sum(i in Tasks)(r[i] * bool2int(b[i])) <= n
```

A better decomposition and flattener would avoid introducing integer variables to represent the constraint, and use direct FlatZinc pseudo-Boolean constraints.

To support this we need FlatZinc constraints

```
predicate bool_lin_eq(array[int] of int: as,
                      array[int] of var bool: bs, var int: x);
predicate bool_lin_le(array[int] of int: as,
                      array[int] of var bool: bs, var int: x);
```

2.6 Recursion

Recursion is necessary to build complex Boolean decompositions of constraints occurring in MiniZinc. The simplest approach to supporting recursion would be to simply allow it, and if the recursion is non-terminating then detect this as a stack overflow and fail the translation process. This is probably good enough in practice. A stronger approach would try to detect loops, and be far more complex.

The addition of recursion will likely require new "list" built-in functions to help support common programming idiom such as:
`function $T: head(array[int] of $T);`
`function array[int] of $T: tail(array[int] of $T);`
`function array[int] of $T: cons($T, array[int] of $T);`
and the corresponding `var` versions. Note that all of these functions simply execute during translation, since the array lengths must always be fixed, so they don't add a burden to FlatZinc solvers.

2.7 Arrays

2d + 3d The translation of array lookups in MiniZinc produces index calculations which require domain consistent propagation to give the best performance. We could allow FlatZinc solvers to directly support 2D and 3D arrays to avoid these problems. The requires changing the `mzn2fzn` translation of arrays to use

```
predicate array_int_element2d(int: m, var int: i, var int: j,
                    array[int] of var int: a, var int: x) =
          x = a[(i-1) * m + j];
```

and

```
predicate array_int_element3d(int: m1, int: m2,
                    var int: i, var int: j, var int: k,
                    array[int] of var int: a, var int: x) =
          x = a[((i-1) * m1 + j-1)*m2 + k];
```

and similarly for `float`, `bool` and `set` so a solver can support them natively. The default decomposition could be above, or make use of a specialized domain consistent version of `int_lin_eq` e.g.

```
predicate array_int_element2d(int:m, var int:i, var int:j,
                              array[int] of var int: a, var int:x) =
          assert(m >= 1, "size of row m in element2d must be positive",
          assert(lb(j) >= 1/\ ub(j) <= m,
                 "range of index j in element2d should be within 1..m",
          let { int: l = (lb(i)-1)*m + lb(j),
                int: u = (ub(i)-1)*m + ub(j)
                var l..u: ij
          } in
                index_calc(m,i,j,ij) /\
                array_int_element(ij, a, x)
          ));
```

where `index_calc(m,i,j,ij)` asserts $ij = (i-1) * m + j$.

Note that we still restrict the type of arrays in FlatZinc *to 1d!*

Even without direct FlatZinc support for 2d and 3d arrays the translation for partially fixed lookups could be improved by partially evaluating it. For example, the expression $x = a[i, 2]$, where a is a 3×3 array $[|1, 2, 3|4, 5, 6|7, 8, 9|]$, is translated to

```
constraint int_lin_eq([-1,3],[ij,i],1) :: domain;
constraint array_int_element(ij,a,x);
```

A better translation would be

```
constraint array_int_element2d(3, i, 2, a, x);
```

or even without 2d arrays

```
constraint array_int_element(i, [2, 5, 8], x);
```

Another example for an opportunity for partial evaluation of lookups during flattening: the constraint $x = a_1[y + 3]$ should be turned into the constraint $x = a_2[y]$ where a_2 is a_1 with an index offset of 3.

Arrays with unknown length It would be nice to support declarations of arrays with unknown length, so that the length can vary from the data file. For example

```
array[int] of int: numbers;
numbers = [1,5,8,3,2];
```

Of course this should not be possible for arrays that are not initialized, but we could allow

```
array[int] of var int: numbers;
numbers = [1,x,8,3,_,2];
```

These should be supported even if the value is in a separate data file.

3 Language Design

There is significant potential for increasing the expressiveness of MiniZinc even without burdening the FlatZinc solver further. Some directions to investigate are:

3.1 Floats

None of the main FlatZinc solver provide a great deal of support for `float` variables so this is under-explored in MiniZinc. We almost certainly need to think about new functions like *sinh*, *cosh*, etc and we have to deal with floating point inaccuracies.

Ideally we should have an interface from FlatZinc to non-linear solvers such as Baron, or at least to quadratic programming solvers such as CPLEX and Gurobi.

3.2 Richer comprehensions

It would be preferable to be able to mix generators and where clauses in comprehensions. This can improve readability as well as make efficient flattening easier. For example:

```
[ a[i] + a[j] + a[k] | i,j in 1..n where i < j,
                       k in i..j where k mod 2 = (i+j) mod 2 ]
```

This would require a substantial change to parsers etc.

The efficiency aspect could be achieved by having the flattener analyse the comprehension before unrolling: for each `where` conjunct it could establish the earliest position in the sequence of generators at which that conjunct can be evaluated.

Variables in generator expressions On a related note, many times it would be convenient to have generator expressions controlled by variables. For example:

```
var set of 1..10: s
constraint forall(i in s)(x[i] >= x[i+1])
```

For some array functions this can be translated to the equivalent

```
var set of 1..10: s
constraint forall(i in 1..10)(i in s -> x[i] >= x[i+1])
```

but this is not obvious for other functions such as `min`, `max`, and certainly not clear for globals such as `alldifferent`. We could support it for comprehensions for `forall`, `exists`, `sum`, `product` if it was seen as worthwhile.

3.3 Sets in types

Sets are currently not always directly usable in types, whereas often this would be desirable. For example

```
let { var index_set(a): x } in E
```

currently needs to be formulated as

```
let { set of int: S = index_set(a), var S: x } in E
```

Another example: given an array of sets such as `A = [ 1..3, 1..4, 1..5, 1..6 ]`, it should be correct to write

```
var A[2]: x;
```

but this needs to be stated as

```
set of int: S = A[2];
var S: x;
```

3.4 Soft constraints

Clearly we want to extend MiniZinc to support soft constraint declarations, and ideally be able to translate soft constraint to optimization objectives in MiniZinc. See the proposal in the proceedings.

3.5 Symmetry Declarations

We can use pseudo-constraint definitions to write symmetry declarations in models, *or* we could extend the language to allow symmetry declarations. Symmetry declarations as pseudo constraints can be handled specially by the FlatZinc backend, or simply translated to static symmetry breaking constraints if not handled (although we have to be careful that the static symmetry breaking is compatible).

As an example

```
predicate var_symmetry(array[int] of var int: x) =
          let { int: l = min(index_set(x)),
                int: u = max(index_set(x)) } in
          forall(i in l..u-1)(x[i] <= x[i+1]);
```

One question is whether the language itself is strong enough to support these. For example if we have a variable pair swap symmetry like $(x_i, y_i) \leftrightarrow (u_i, v_i)$ over arrays of variables, can we express this using MiniZinc types.

Proposal 3. A set of global constraints for symmetry declaration with good static symmetry breaking should be added to the MiniZinc globals library.

3.6 Type Extensions

Clearly we could extend MiniZinc to include richer types, and indeed we can use the syntax and semantics of Zinc to define these extensions. None of the type extensions we discuss below needs to change the FlatZinc interface, but once we introduce a type extension there may well be pressure to do so.

The question is what do we want and how important is each one:

enumerated types : The simplest extension, since it amounts to a source to source transformation, although complexities arise if we wish to handle enumerated types defined in data files and separate compilation.

tuples : The most obvious missing data structure is tuples. Their lack makes some things hard or inefficient to model. The key problem here is that the type system is now infinite.

records : Not that much further than tuples, but useful for complex models.

nested arrays : Useful for sparse matrices.

3.7 Annotations

Annotations are a fairly flexible way of passing information from the model to the FlatZinc solver, **but** the way they are handled by rewriting is naive and unsuitable for a number of useful annotations.

Questions about annotations include:

- Can we annotate all the places we need to, should we be able to annotate arbitrary subterms, e.g. `constraint x + (y :: bounds) = z;`?
- Priority of the annotation operator, given its typical use. For example, we currently need to write `(x = y + 3) :: domain` to annotate the equality constraint, whereas it might just be `x = y + 3 :: domain`
- What standard annotations should be added to MiniZinc and FlatZinc?
- How can we control better how annotations are treated by flattening? Are there "classes" of annotations in terms of flattening treatment?

3.8 Output

Originally FlatZinc solvers had to support the output statements in a MiniZinc model, but this was removed in later versions of MiniZinc by simply requiring solvers to return a solution stream in MiniZinc data format (`.dzn`). Hence output processing is now completely isolated from the solvers. We could rewrite the output processing part of MiniZinc completely.

Apart from the simple changes discussed above, we could include a complete different output language, since sometimes the functional processing style is unnatural, particularly without recursion.

3.9 Functions

Allowing the user to write functions can significantly improve the expressiveness of MiniZinc models but raises a whole host of questions about partiality. The language extension is not difficult in the sense that we can simply hijack the syntax from Zinc. But this would mean that translation would be considerably more complex.

On a related note, at present MiniZinc only supports the relational semantics. Should we consider allowing translations that define the strict semantics and the Kleene semantics [20]?

3.10 Syntax

MiniZinc uses some unique syntax which makes it perhaps harder for uptake by new modellers. Some possible changes to syntax might be:

- Remove `==` (and just allow `=`).
- Replace `/\` and `\/` by `&&` and `||`, or `and` and `or`.
- Remove the `constraint` key word.
- Assume `solve satisfy` if no solve goal is explicitly given.
- Omit the curly brackets in `let` expressions, as in `let var T: x in E .`
- Interpret `<=`, `<` on arrays as the corresponding lexicographic ordering constraints.
- Change the syntax for comprehension expressions. This would remove the requirement for some parentheses but seems only practical for builtins like `sum`, `exists`, `forall` where we know the precedence.
- Currently the grammar is not very easy to parse, we could make it LALR(1) but this would require significant changes.

4 MiniZinc Toolset

The MiniZinc toolset includes: the translator from MiniZinc to FlatZinc, `mzn2fzn`; the translator of solution to data file format, `soln2dzn`; the translation of solutions to MiniZinc output, `soln2out`; as well as translator to and from FlatZinc and FlatZinc XML format; and finally the standalone interpreter

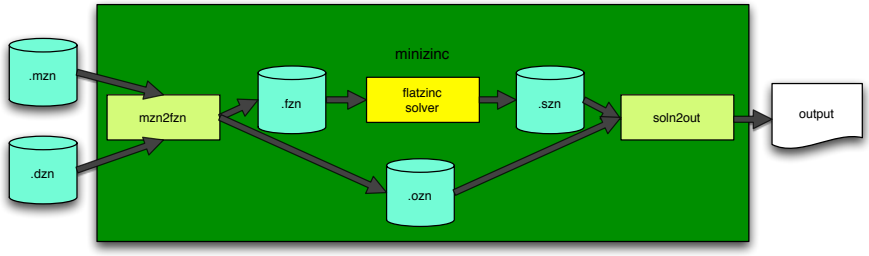

Fig. 1. The MiniZinc toolchain.

`minizinc` that marshals these tools for a simple one line usage. The tool chain is illustrated in Figure 1. The model (`.mzn`) and data files (`.dzn`) are joined and flattened to create a FlatZinc file (`.fzn`) as well as an output file (`.ozn`) recording the output statement and variables required for it. The FlatZinc solver creates a solution stream (`.szn`) and this is combined with the output file to create the output of the system.

4.1 Runtime options

When running a FlatZinc solver (particularly through the tool `minizinc`) we want to be able to control the solver using runtime flags. We should try to standardize these as far as possible.

- What runtime flags do we need?
- Can we support arbitrary other flags to be passed through tools such as `minizinc`?

Proposal 4. If one person would carefully go through the runtime flags for a reasonable subset of the solvers supporting FlatZinc we should be able to get a reasonable proposal for standardizing them fairly easily.

4.2 Statistics

In order to compare solvers, or indeed just to compare models, it is important to have meaningful statistics extracted from the solver. We should standardize runtime flags for FlatZinc solvers to control statistics output, as well as the statistics output format. Some issues are:

- What should be the behaviour of `flatzinc -s`?
- Important statistics like: `time`, `fails`, `choices`, `nodes`, `backtracks`, `memory` should be formalized and be controllable from run time flags.
- How do we allow arbitrary different statistics for different solvers, e.g. `nogoods`, `recomputations`?
- How do statistics work with all solutions?

4.3 FlatZinc

FlatZinc has remained fairly stable over the lifetime of MiniZinc, the main change being addition and deprecation of predicates. Should we consider a very different style language that is not flat? FlatZinc models are often extremely large, even for some easy problems. Should we try to make FlatZinc more concise? (One such step may just be the omission of the `constraint` keyword; see 3.10.)

4.4 Visualization

Visualization of solving and search is vital for CP solvers for understanding performance problems.

We have extended MiniZinc to support the CP-Viz [23] visualization approach, and added annotations to the language which allow this to be pushed through to the FlatZinc solvers.

We certainly need more work to make a fuller integration. The key question is how do we motivate solver developers to add support for CP-Viz and how do we create a uniform framework for this that is simple to support, or support partially.

4.5 Integrated Development Environment

An integrated development environment (IDE) is an important tool for the easy takeup and usage of new software. Given that constraint programming is a challenging field to enter in any case, a good IDE is very important for making MiniZinc accessible.

An integrated development environment (IDE) was developed for G12, including MiniZinc previously, but this was not developed at the same lab as the rest of the components, and is no longer funded for support.

- How can we create a new IDE quickly and cheaply with minimal support requirements?
- What features would we like to have in an IDE, beyond the minimal set?
- Can we get community interest to support such a tool?

4.6 Integration

At present MiniZinc is a standalone language and toolset, and integration into other systems is only possible through manipulating text files. This makes the takeup of MiniZinc by industry problematic. The `libmzn` proposal in this workshop has a comprehensive suggestion to overcome these problems.

Some alternate paths should also be considered:

- An XML format for MiniZinc and `.dzn` files, as well as tools to convert from text to XML
- Database interfaces for the current toolset, or at least a standard way of converting e.g. `.csv` files to `.dzn`.

4.7 Alternates

Nothing is stopping someone else creating a MiniZinc to FlatZinc translator and indeed `mzn2fzn` is completely open source (though written in Mercury which may not be too accessible). It isn't that difficult and indeed there was a paper that described a more direct translation from MiniZinc to `Gecode` [24].

Different translators would certainly provide impetus to improve translation, although perhaps the `libmzn` approach described elsewhere in this workshop provides a better approach.

5 Outreach and Organization

The eventual aim is for MiniZinc to be a standard for constraint programming modelling, and preferably for other combinatorial optimization technologies as well. How do we get there?

5.1 Documentation

While we believe the FlatZinc documentation is reasonable, the MiniZinc documentation is part of the Zinc documentation, and this should clearly we addressed. It would also be good to have more detailed documentation for solver developers which show exactly the steps needed to go through to support MiniZinc.

What other documentation is missing?

5.2 Problem Repository

There are a large collection of MiniZinc models available in various places. Ideally we should have a publicly accessible repository that

- Contains a clear description of the problem and links to related material on the web
- Is easy to add new problems and instances to
- Allows testing a solver on a problem/instance or large sets of problem/instances easily.

We have a wiki (but now you need to have an account to edit it) which has not seen much external buy in.

Should we insert this within CSPlib [25]?

5.3 Forum

At present the only information stream for MiniZinc is the Minizinc developers mailing list, and a Mantis bug reporting site [26]. We need a forum that is accessible and archived. We should at least consider MiniZinc developers and MiniZinc users forums. Are there other communication mechanisms we should put in place?

5.4 Competition

The MiniZinc Challenge is an important tool to motivate solver writers to support MiniZinc, **but** often solvers only enter once, and don't reenter in later years.

- How can we improve the participation retention in the contest?
  - More categories and sub-categories?
  - Specialised comparison: CP/MIP/SAT-SMT ?
- How can we improve the collection of problems? We struggle to find 10 new problems, particularly this year when no solver writers took the opportunity to add a benchmark that favors their system.
- Is the comparison fair enough, is their a better scoring system?
- How can we attract more new entrants? What do solver writers perceive as the barrier to entry in the Challenge?

On another note, we would be very happy if someone wants to volunteer to run the Challenge for a year, and we promise to submit many example problems!

5.5 Learning MiniZinc

Obviously we believe that MiniZinc is a low barrier entry point for learning constraint programming. We should endeavour to support those learning MiniZinc as well as possible.

There is a lengthy tutorial as part of the distribution. Should we create a shared online tutorial document?

In Melbourne a number of us share the same course material for teaching MiniZinc, and the material is available from our web sites. Should we consider a shared online repository for teaching material?

5.6 Governance + Resourcing

All decisions about MiniZinc are currently made by the G12 team, although we try to discuss things first with MiniZinc developers (those whose solvers support MiniZinc).

Clearly the governance of MiniZinc needs to be by the contributors. Should we move to a separate governance structure, an association or non-profit organization? Is there enough buy in for this? Would the creation of this structure encourage more contribution from others?

5.7 Standards

We are not in any hurry, but finally what are the processes to make MiniZinc a standard? What should the timeline be? What standards organization?

6 Conclusion

We have discussed a number of future directions for MiniZinc and FlatZinc: some of them quite concrete and simple enough to add to the current system; some of them requiring major changes to the system. In order for wide adoption of MiniZinc the constraint programming community must see it as valuable. We believe we have done a reasonable job on making it attractive for solver writers to support MiniZinc. The real question here is how can we motivate the CP community to use MiniZinc and contribute to MiniZinc development.

References

1. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: Minizinc: Towards a standard CP modelling language. In Bessiere, C., ed.: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. Volume 4741 of LNCS., Springer-Verlag (2007) 529–543
2. : Minizinc + flatzinc web site. http://www.g12.csse.unimelb.edu.au/minizinc/
3. : Cplex. www.ibm.com/software/integration/optimization/cplex-optimizer/
4. : Gurobi. http://www.gurobi.com/
5. : Osi cbc. http://www.coin-or.org/Cbc/cbcuserguide.html
6. : Minisat. http://minisat.se/MiniSat.html
7. Schulte, C., Lagerkvist, M., Tack, G.: Gecode. http://www.gecode.org/
8. Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2006)
9. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Procs. of Programming Langauges, Implementations, Logics and PRograms, PLILP97. Volume 1292 of LNCS., Springer (1997) 191–206
10. : Jacop. http://jacop.osolpro.com/
11. Huang, J.: Universal Booleanization of constraint models. In Stuckey, P., ed.: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming. Volume 5202 of LNCS., Springer (2008) 144–158
12. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In Perron, L., Trick, M., eds.: Proc. of CPAIOR 2008. Volume 5015 of LNCS., Springer (2008) 6–20
13. : fzn2smt. http://ima.udg.edu/Recerca/ESLIP/fzn2smt/index.html
14. : B-prolog. http://www.probp.com/
15. Metodi, A., Codish, M., Lagoon, V., Stuckey, P.: Boolean equi-propagation for optimized SAT encoding. In Lee, J., ed.: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming. LNCS, Springer (2011) to appear
16. G12 Team: The minizinc wiki. http://www.g12.csse.unimelb.edu.au/wiki/doku.php
17. Stuckey, P.: Peter Stuckey's home page. www.unimelb.edu.au/~pjs
18. Kjellestrand, H.: My minizinc page. http://www.hakank.org/minizinc/
19. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1-3) (1996) 17–64
20. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In Gent, I., ed.: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Volume 5732 of LNCS., Springer-Verlag (2009) 367–382

21. Beldiceanu, N.: Global constraints catalog. http://www.emn.fr/z-info/sdemasse/gccat/
22. Feydy, T., Somogyi, Z., Stuckey, P.: Half-reification and flattening. In Lee, J., ed.: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming. LNCS, Springer (2011) to appear
23. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., Carlsson, M.: A generic visualization platform for CP. In: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming. Volume 6308 of Lecture Notes in Computer Science., Springer (2010) 460–474
24. Cipriano, R., Dovier, A., Mauro, J.: Compiling and executing declarative modeling languages to Gecode. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the 24th International Conference on Logic Programming. LNCS, Springer (2008) 744–748
25. : Csplib. http://www.csplib.org/
26. : G12 bugs database. http://bugs.g12.csse.unimelb.edu.au/login_page.php