An Abstract Machine Model for MiniZinc

Jip J. Dekker (student)[1,2], Andreas Schutt[2], Maria Garcia de la Banda[1], Graeme Gange[1], Peter J. Stuckey[1,2], and Guido Tack[1,2]

[1] Monash University, Melbourne, Australia
{jip.dekker,graeme.gange,maria.garciadelabanda,peter.stuckey,guido.tack}@monash.edu
[2] Data61, CSIRO, Melbourne, Australia
adreas.schutt@data61.csiro.au

Abstract. The use of meta-heuristic algorithms, such as Large Neighbourhood Search, Lexicographic Search, and Interactive Search, has proven to be very successful; however, the support for the use and development of these algorithms within constraint modelling languages is highly limited. In this paper we introduce a machine model for the incremental evaluation of constraint modelling languages. The model matches the functionality of modern modelling languages, such MiniZinc, Essence, AMPL and OPL, and its inherent incremental nature lends itself extremely well for defining meta-heuristics algorithms.

Keywords: Constraint Modelling, Meta-Heuristics, Incremental Evaluation, Interpreter, MiniZinc

1 Introduction and Related Work

Constraint modelling languages, such as MiniZinc [8], Essence [5], AMPL [4] and OPL [6], are popular tools to formulate optimisation problems. They are relatively easy to learn, abstract from the solver's internal mechanism, and even allow the user to switch between different solvers. However, solving optimisation problems has always been complex and recent developments have seen a massive increase in the use of so called *meta-heuristics*. These general search algorithms guide solvers in their exploration of the global search space by repeatedly selecting subsets for the solver to explore. Popular examples of these algorithms are Large Neighbourhood Search (LNS [10]), Lexicographic Search [7], and Interactive Search [9].

Various approaches have been proposed to add the ability to define meta-heuristics to constraint modelling languages, including:

- Search Combinators [12], which add programmable search and meta-heuristics to solvers through an interpreter within the solver's search mechanism.
- Solver Independent LNS [2], which allows the modelling of single level meta-heuristics in MiniZinc and makes them available in extended solvers.
- MiniSearch [11], which provides programmable meta-heuristics through an interpreter language within a MiniZinc model.

Each of these solutions operates at a different stage during the solving of an optimisation problem: Search Combinators are directly implemented within the solver; Solver Independent LNS uses a combination of compilation and extensions of the solver's propagation engine; and MiniSearch repeatedly re-compiles the model and calls the solver. Despite these differences, most meta-heuristics have the following structure: They first search for a solution, add or remove constraints from the problem (based on the solution), and repeat. If a meta-heuristic is to be specified within a modelling language, then it means that the constraints added in the second step are defined in the modelling language. Hence, we can compile these constraints to the solver level in a generic form, so that the solver can instantiate them for each new solution found. This is the approach taken by Solver Independent LNS. Although this method requires the least amount of re-computation, it does require all constraints that will be used to be in the model and the activation logic of these constraints is limited. Alternatively, we have to re-compile the model for each solution. This allows us to dynamically add or remove any constraint during the solving of the optimisation problem, but does require more computational work. In this paper we optimise the re-compilation approach by making it *incremental*, i.e., only processing the new, added constraints, without re-compiling the entire original model.

This paper introduces a machine model for the evaluation of constraint modelling languages that is fully incremental. In this paper the process is described in terms of MiniZinc; however, the same principles would apply to other constraint modelling languages. After the reader is introduced to the traditional compilation process of MiniZinc, Section 2, we describe an overview of new the evaluation process, Section 3, and then delve into important details regarding dependency tracking, Section 4, and common sub-expression elimination, Section 5. The choices explained in these sections will allow for the incremental evaluation of MiniZinc through the use of backtracking. The technical details of which are explained in Section 6.

2 Background

A MiniZinc *model* is combined with data to form a MiniZinc *instance*. Generally, solvers cannot solve MiniZinc instances directly. Thus, an equivalent *flat* constraint model, referred to as *FlatZinc*, is generated containing only constraints supported by the solver. A constraint model is considered flat when all high level constructs (e.g., loops, nested expressions, predicates, functions, enum-types) have been eliminated. The full process is depicted in Figure 1.

A common misconception is that MiniZinc is a simple rewrite system that rewrites constraints until it reaches flat state with constraint accepted by the solver; however, MiniZinc has to take special care to produce a constraint model that will perform well in the solver. For example, all expressions that do not contain decision variables are evaluated immediately, the results of an identical expressions are reused, and the domains of decisions variables are tightened when possible.

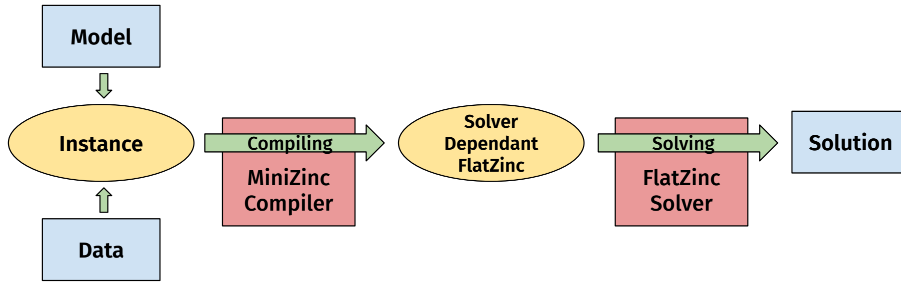

Fig. 1: A diagram representing the process that a MiniZinc instance undergoes to arrive at a solution.

3 Overview of the Machine Model

Figure 2 shows the newly proposed process for solving a MiniZinc instance. In this approach the compilation process from MiniZinc to FlatZinc is split into two stages: (1) the compilation of the MiniZinc instance into a bytecode program, and (2) the evaluation of the bytecode program to the solver dependant FlatZinc. Although the compilation to a bytecode program brings its own complications, we will focus on the second step and assume that there exists a compiler to produce a bytecode program.

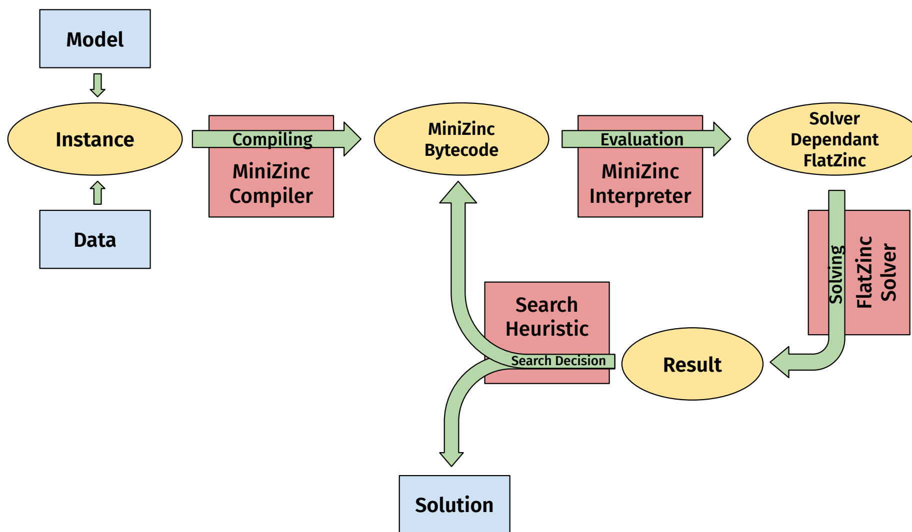

Fig. 2: A diagram representing the newly proposed process that a MiniZinc instance would undergo to arrive at a solution.

We define *incumbent* FlatZinc as a list of *calls*, such as `var int: i1 = count(X, y);`. Each call consists of a domain, an identifier, a function identifier, and a list of arguments. For simplicity, we will assume that arguments can only be numbers, identifiers, or lists of numbers and/or identifiers. The evaluation of a bytecode program can be seen as a rewriting system. In each rewriting step one call in the incumbent FlatZinc is evaluated. If the call is supported by the target solver, then it will remain in the model; otherwise, the call will be rewritten into (i.e., replaced by) a set of new calls according to the given bytecode. In the replacement there must be a call that is semantically equivalent and uses the identifier of the original call.

Take for example the following MiniZinc code which constrains $|a - b| = 2$ using a custom predicate:

```
1 predicate two_apart(var int: i, var int: j) =
2   i + 2 = j \/ j + 2 = i;
3
4 var 1..5: a;
5 var 3..6: b;
6
7 constraint two_apart(a, b);
```

The initial incumbent FlatZinc for this instance would include the calls:

```
1 var int: a = mk_var(1, 5);
2 var int: b = mk_var(3, 6);
3 true: i1 = two_apart(a, b);
```

where `mk_var` is the identifier of a solver built-in function used to create a fresh variable. Note that the domain of these calls is tightened during the optimisation of the incumbent FlatZinc. Evaluation will not rewrite these two calls. The call to `two_apart` will be rewritten by evaluating the bytecode generated for the `two_apart` predicate. This will replace the call by two linear constraint calls and a call for the disjunction between them, in accordance to the predicate definition given in the MiniZinc code. This results in:

```
1 var int: a = mk_var(1 ,5);
2 var int: b = mk_var(3, 6);
3 true: i1 = bool_or(i2, i3);
4 var bool: i2 = int_plus(a, 2, b);
5 var bool: i3 = int_plus(b, 2, a);
```

In this paper, we will not go into the details of how exactly the bytecode is structured or evaluated. For the rest of the paper, we will simply assume that any call to a function (or predicate) that has a definition in MiniZinc can be rewritten, while any other call that does not have such a definition is a solver built-in.

Note that the output of any rewriting step results in valid incumbent FlatZinc. Some of the calls may already be solver built-ins, while others still require

evaluation. At any point we can therefore propagate any call for which an internal propagator is available. This can improve the incumbent FlatZinc by for example, fixing some variables to constants, and constraints becoming implied (and therefore removed from the incumbent FlatZinc) or simplified. Not all calls in the incumbent FlatZinc will have a propagator available; however, the partial propagation of a constraint model is valid for the full model. Note that the internal constraint solver we use for this optimisation step can only propagate constraints, it does not perform any search.

An important design decision is *when* to perform this partial constraint propagation. While we plan to experiment with different approaches, a general rule could be to propagate simple constraints during the evaluation, as they could already provide information that would simplify the evaluation of future calls. More complex constraints may be delayed until the end of the evaluation process, or even left for the solver.

4 Dependency Tracking

During the evaluation of a bytecode program, the interpreter might evaluate calls that prove to be unnecessary later in the process due to propagation or to the order in which the calls are evaluated. Take for example the following MiniZinc code which constructs an array containing the results of divisions between different variables and forces the second element to be 3:

```
1 array[1..4] of var 0..10: x;
2 constraint [ x[i] div x[i+1] | i in 1..3 ][2] == 3;
```

Assuming the compiler generates a naive bytecode program, it will evaluate the functional definition for each division and then use an element constraint to force the second division to be equal to 3. Note that this also could have happened if value 2 was yet unknown, but is fixed through propagation. The incumbent flatzinc would look like this:

```
1 var int: x1 = mk_var(0, 10);
2 var int: x2 = mk_var(0, 10);
3 var int: x3 = mk_var(0, 10);
4 var int: x4 = mk_var(0, 10);
5 var int: i1 = int_div(x1, x2);
6 var int: i2 = int_div(x2, x3);
7 var int: i3 = int_div(x3, x4);
8 var int: i4 = element([i1,i2,i3], 2);
9 true: i5 = int_eq(i4, 3);
```

Simplification of the `element` call results in the unification if `i4` and `i2`. At this point the calls `i1` and `i3` are no longer necessary, and propagating them could waste valuable time.

To eliminate these unused calls, we need to keep track of the *liveness* of each identifier: in general, an identifier is live if it is used as an argument to another call. A common way to track liveness is by using reference counting, which keeps track of the number of times an identifier is used as an argument to other calls. Once this number drops to zero, the identifier is known to be dead and its call can be removed from the incumbent FlatZinc. In MiniZinc, there are two special cases to consider. An identifier that is required for the output of the model should never be removed. Likewise, an identifier whose domain is *binding*, i.e., actually constraining the result of its call, must also not be removed.

While this scheme, used in the current MiniZinc compiler, looks enticing for its simplicity, basic reference counting cannot keep track of the dependencies in complex expressions potentially resulting in large parts of the FlatZinc not being detected as dead.

Consider for example a version of the incumbent FlatZinc above where the `div` operator is no longer simply rewritten to `int_div`, but to a redefinition which ensures the divisor is not zero:

```
1 function int: 'div'(int: x, int: y) = let {
2   constraint y != 0;
3   var int: yy = if y=0 then 1 else y endif;
4   var int: r;
5   constraint int_div(x, yy, r);
6 } in r;
```

Using this definition every division would result in multiple calls; for example, the first division resulting in variable `i1`, on line 5, would become:

```
1 var int: i1 = mk_var(0, 10);
2 true: i11 = int_ne(x2, 0);
3 var int: i12 = mk_var(1, 10);
4 var bool: i13 = int_eq(i12, 1);
5 var bool: i14 = int_eq(i12, x2);
6 true: i16 = bool_clause([i13, i11], []); % (not i11) -> i13
7 true: i15 = bool_clause([i14], [i11]); % i11 -> i14
8 true: i17 = int_div(x1, i12, i1);
```

The division no longer functionally defines the identifier `i1`. When `i1` becomes unused its count does not drop to zero. It is still used in the `int_div` call. Even if one applies special cases for identifiers where a relational constraint can be rewritten into a functional form, it would still not fully solve the problem. For example `i12`, would no longer be necessary, but because its value is defined by the combination of two separate calls it is impossible to determine that it can be removed using reference counting alone.

To solve this problem we introduce the concept of *dependant calls*. For every call we keep track of all other calls that are created to help define it. The evaluation of a call will bind all calls generated to the call with the same identifier. This means that the evaluation of a call will now no longer return a *list* of calls, but a

tree of dependent calls. The incumbent FlatZinc will thus be represented as a list of these trees, which we call a *hedge*. The hedge structure, shown by indentation, of the previous example would look like this:

```
1 var int: i1 = mk_var(0, 10);
2     true: i11 = int_ne(x2, 0);
3     var int: i12 = mk_var(1, 10);
4         var bool: i13 = int_eq(i12, 1);
5         var bool: i14 = int_eq(i12, x2);
6         true: i16 = bool_clause([i13, i11], []);
7         true: i15 = bool_clause([i14], [i11]);
8     true: i17 = int_div(x1, i12, i1);
```

The structure of the hedge can be used to increase the effectiveness of the reference counting. Instead of counting all occurrences of an identifier, we only count the occurrences within calls that do not occur within its tree of dependant calls. We also specify that all dependant calls have their reference count increased by 1. If the reference count of a call reaches zero, then it is removed and all its dependants have their reference count decreased by 1. If their reference count also reaches zero, then the process is repeated recursively; however, if they have a positive reference count, then the call is promoted upwards in the tree. Using these reference counting rules, all calls in the example can be removed once the reference count of `i1` reaches zero.

5 Common Sub-Expression Elimination

The main goal for the evaluation of the bytecode program is to create FlatZinc that a solver is able to solve as fast as possible. An issue that can cause unnecessary work for a solver is the duplication of identical constraints in the FlatZinc. Not only does this result in extra work for the solver propagating the duplicates, it can lead to much larger search trees if the solver cannot detect the equality between two duplicated expressions. To solve this problem modern constraint modelling languages use common sub-expression elimination (CSE) [1]. In a MiniZinc model constraints express relationships between variables. If the same, or an equivalent, constraint is placed multiple times on the same variables, it expresses the same relationship multiple times. The result of the evaluation of a constraint can thus be reused.

During the evaluation of a bytecode program CSE can be used for the evaluation of all calls, except for a selected set of functions which do not describe a relationship, but are instead operational calls to the solver or interpreter. An example of such a function is `mk_var`, which requests the creation of a fresh variable. For the evaluation of all other calls CSE would function as a lookup before the evaluation of the call. If a call with the same function identifier and arguments was previously evaluated, then we can reuse the stored result; otherwise, the call is evaluated normally, and its information and result are stored for future lookup operations.

To increase the efficiency of CSE during evaluation it should be aware of *reifications*. A reified constraint associates a truth value with a constraint. Take for example the call `int_le(a, b)` and its reification `int_le_reif(a, b, r)`, which constrain $a \leq b$ and $r \iff (a \leq b)$ respectively, during the evaluation of a bytecode program containing both calls, only the first constraint needs to be added to the incumbent FlatZinc and, afterwards, `r` is known to be `true`. Reification aware CSE has high potential in removing unnecessary reified constraints, as they add both extra decision variables and constraints.

In our new bytecode interpreter we consider not just constraints and their reified versions, as the current MiniZinc compiler does, but also half-reified versions [3], which imply a constraint, and their exact negations. We thus consider the following *contexts* for every function:

- Root context (e. g., `int_le(a, b)`)
- Negated root context (e. g., `int_gt(a, b)` in case of `int_le(a, b)`)
- Reified context (e. g., `int_le_reif(a, b, r)` in case of `int_le(a, b)`)
- Negated reified context (e. g., `int_gt_reif(a, b, r)` in case of `int_le(a, b)`)
- Half-reified context (e. g., `int_le_imp(a, b, r)` in case of `int_le(a, b)`)
- Negated half-reified context (e. g., `int_gt_imp(a, b, r)` in case of `int_le(a, b)`)

To manage our CSE structure, an ordering of the context can now be established where root context > reified context > half-reified context. Only the result of the highest context needs to be stored for our lookup operation, which includes the context used. For root context the result is a fixed Boolean value (true or false), while for reifications the result is the reified variable (`r`). Whenever a lookup operation finds a match it compares the current evaluation context to the context in the result. If the context level in the result is equal or higher, then the previous result can be used and if only one of the contexts of the result and the evaluating context is negated, then the result that was found will need to be negated. Otherwise, the evaluation proceeds as normal and the previous result stored in the CSE structure is replaced. Because of our dependency tracking, explained in Section 4, the old result and all its dependencies can be correctly removed. This ensures that the order in which the bytecode is evaluated does not effect the resulting FlatZinc.

When bytecode programs get bigger, the amount of calls that will be evaluated rises quickly. It is therefore important to consider the cost of CSE. The memory used and time taken for processing a CSE request can be limited to a maximum size or its functionality can be limited to a set of functions.

6 Backtracking Interpreter

We can provide incremental functionality during the evaluation process by adding backtracking functionality. At each point during the evaluation, we allow the user to (1) push a choice point, marking the current moment in the evaluation; (2) add a call, allowing the creation of new constraints and variables; and (3) pop a choice point, returning the state to the previous choice point and removing

this one. Due to the nature of our evaluation process, the ability to add a call to the incumbent FlatZinc is trivial. The incumbent FlatZinc can be extended with the new call (at the top level of the hedge) at any point in time and further evaluation can be triggered. The use of choice points will require the use of trailing: Every change made to the incumbent FlatZinc after a choice point is pushed is recorded; When a choice point is popped, the changes that have been recorded since the previous choice point are reverted.

Changes to incumbent FlatZinc can be either the addition of a call, the removal of a call, or the change in the domain of a call. Tracking the addition of calls can be done by storing the position of the latest (top level) call added to the incumbent FlatZinc at every choice point. If the choice point is popped, then all calls added after the stored position are removed. The propagation to optimise the incumbent FlatZinc could change the domain of calls. If the propagation changes a call that existed before the last choice point, then the original domain has to be recorded. This domain will be reverted when the choice point is popped. (Note that as the same domain can change multiple times, either the recording structure has to account for this, or the recorded changes have to be reverted in order). Finally, propagation and CSE can remove calls from the incumbent FlatZinc. This happens for two reasons: either a stronger constraint was added eliminating the need for the call or, a call was added which is more powerful. For example, a reification gets added and replaces the need for a half-reification. In the former case, the call and its position are added to a list of removed calls. When we revert to the previous choice point it is added back. In the latter case, a call is not only removed but also replaced. In this case the removed call and its position are stored and its name will be an alias to the new call. Any references to the old call will therefore use the definition of the call which replaces it. If the choice point is popped, then the call is restored to the position of the alias.

To take full advantage of the incremental evaluation of a bytecode program the solver should be implemented in a similarly incremental fashion. This would avoid the repeated startup costs of solvers and would allow solvers to keep the data learned during its execution. An incremental interface for a solver can be supported at multiple levels. A full incremental solver would be required to allow the addition of constraint and variables (after solving has started) and the reverting to an earlier state; however, even partial support for these methods might avoid the need to restart the solver in some cases.

7 Conclusion and Outlook

In this paper we have shown a new machine model for the evaluation of MiniZinc that is fully incremental. The incremental nature of the evaluation allows the user to define meta-heuristics without any restarts during the evaluation of the program. An implementation of the machine model presented in this paper is currently under development. We hope to distribute a full compilation suite according to this machine model within the MiniZinc bundle in the future.

References

1. Araya, I., Neveu, B., Trombettoni, G.: Exploiting common subexpressions in numerical csps. In: Stuckey, P.J. (ed.) Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5202, pp. 342–357. Springer (2008), https://doi.org/10.1007/978-3-540-85958-1_23
2. Dekker, J.J., de la Banda, M.G., Schutt, A., Stuckey, P.J., Tack, G.: Solver-independent large neighbourhood search. In: Hooker, J.N. (ed.) Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11008, pp. 81–98. Springer (2018), https://doi.org/10.1007/978-3-319-98334-9_6
3. Feydy, T., Somogyi, Z., Stuckey, P.J.: Half reification and flattening. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 286–301. Springer (2011), https://doi.org/10.1007/978-3-642-23786-7_23
4. Fourer, R., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Duxbury (2002)
5. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008), https://doi.org/10.1007/s10601-008-9047-y
6. Hentenryck, P.V.: Constraint and integer programming in OPL. INFORMS Journal on Computing 14(4), 345–372 (2002), https://doi.org/10.1287/ijoc.14.4.345.2826
7. Jones, D.F., Mirrazavi, S.K., Tamiz, M.: Multi-objective meta-heuristics: An overview of the current state-of-the-art. European Journal of Operational Research 137(1), 1–9 (2002), https://doi.org/10.1016/S0377-2217(01)00123-0
8. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer (2007), https://doi.org/10.1007/978-3-540-74970-7_38
9. Phelps, S.P., Köksalan, M.: An interactive evolutionary metaheuristic for multiobjective combinatorial optimization. Management Science 49(12), 1726–1738 (2003), https://doi.org/10.1287/mnsc.49.12.1726.25117
10. Pisinger, D., Ropke, S.: Large neighborhood search. In: Handbook of Metaheuristics, pp. 399–419. Springer US (2010), https://doi.org/10.1007%2F978-1-4419-1665-5_13
11. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: Minisearch: A solver-independent meta-search language for minizinc. In: Pesant, G. (ed.) Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9255, pp. 376–392. Springer (2015), https://doi.org/10.1007/978-3-319-23219-5_27
12. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. Constraints 18(2), 269–305 (2013), https://doi.org/10.1007/s10601-012-9137-8