Multi-Pass High-Level Presolving

Kevin Leo and **Guido Tack**
Faculty of IT, Monash University, Australia
and National ICT Australia (NICTA) Victoria
{kevin.leo,guido.tack}@monash.edu

Abstract

Presolving is a preprocessing step performed by optimisation solvers to improve performance. However, these solvers cannot easily exploit *high-level model structure* as available in modelling languages such as MiniZinc or Essence.

We present an integrated approach that performs presolving as a separate pass during the compilation from high-level optimisation models to solver-level programs. The compiler produces a representation of the model that is suitable for presolving by retaining some of the high-level structure. It then uses information learned during presolving to generate the final solver-level representation.

Our approach introduces the novel concept of *variable paths* that identify variables which are common across multiple compilation passes, increasing the amount of shared information. We show that this approach can lead to both faster compilation and more efficient solver-level programs.

1 Introduction

Presolving is commonly used in linear and mixed-integer linear programming (LP/MIP) solvers and Boolean satisfiability (SAT) solvers to improve performance. A presolver analyses and improves a problem instance before the actual solving process starts, inferring tighter variable domains, simplifying constraints, and removing variables and constraints that are guaranteed not to contribute to a solution.

A compiler for a high-level constraint modelling language performs similar optimisations when it translates a constraint model to a solver-level program. It will try to compute tight variable bounds, select the most suitable variant of each constraint, and generate a compact model that does not contain unused variables or constraints.

The main difference between a presolver and an optimising compiler for a modelling language is the level of detail available for analysis. A presolver can analyse the entire problem instance, with all variables and constraints present and accessible. In traditional compiler terminology, this would be called *whole program optimisation*. A compiler for a modelling language, on the other hand, typically performs analysis and compilation simultaneously, while it is constructing the solver-level program. The compiler's knowledge of the whole generated program is therefore, inevitably, only partial until it is finished compiling. However, it can use the high-level structure available in the input model, such as global constraints, functional dependencies, and a rich expression language, to perform more powerful inference.

This paper presents techniques for **integrating whole-program optimisation** into compilers for constraint modelling languages. The resulting system combines the advantages of analysing the entire model with the additional inference strength possible due to the high level model structure being available. This is achieved by compiling in *multiple passes*: The initial passes translate the input model into various representations suitable for whole-program analysis. The final pass uses information gained from the earlier analysis, in the form of tightened variable domains and learned variable aliases, to produce a better solver-level program.

Contributions The main technical contribution of this paper is an architecture for multi-pass presolving during compilation of high-level constraint models. Information is communicated from one pass to the next by computing unique variable identifiers called *variable paths* that are stable across different compilations. This enables the compiler to generate one version of a constraint model suitable for presolving, and another version suitable for the target solver. Presolving then takes advantage of the high-level model structure, for instance using strong inference methods such as constraint propagation, even if the target is not a propagation-based solver.

Our experiments show that two-pass, propagation based presolving, can produce significantly better solver-level programs, reduce the numbers of variables and constraints, tighten variable bounds, and lead to improved solve times.

2 Background

This section introduces some concepts used in later sections.

2.1 Constraint Models and Programs

We are concerned with the compilation of high-level constraint *models* to concrete solver-level *programs*. A *model* is a parametric specification of a constrained problem. It contains declarations of parameters and decision variables, as well as constraints expressed in terms of nested expressions, loop

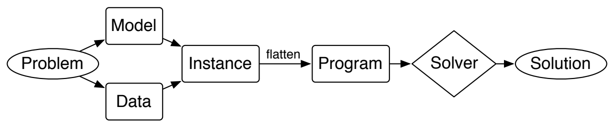


Figure 1: Typical compilation chain

constructs (such as `forall` and `exists` quantifiers, array/set comprehensions), and function and predicate definitions and calls. In addition, models for optimisation problems specify an objective function to be minimised or maximised.

A *program* is a lower-level problem specification that is understood directly by the target solver. It contains variable declarations and constraints, where each constraint is a simple call to a *builtin*, a constraint supported natively by the solver.

Different solvers support different types of variables and constraints. Constraint Programming (CP) solvers typically provide many high-level constraints over variable types such as Booleans, integers, floats, and sets, ranging from simple binary relations to complex global constraints that describe entire sub-problems. This allows a CP solver to operate on a reasonably high-level, compact representation of a problem. CP solvers intersperse *constraint propagation* and *search*. Propagation uses dedicated algorithms for each constraint in a program that remove inconsistent values from variable domains. In this paper, we use constraint propagation as a presolving algorithm.

Solving approaches other than CP often require programs that are more restrictive in their use of variable and constraint types. The language of mixed-integer linear programming (MIP) solvers requires variable domains to take either integer or continuous values from contiguous ranges, while constraints must be linear (in)equalities. Programs for Boolean satisfiability (SAT) solvers are expressed in a similarly restrictive language with only Boolean variables and clause (disjunction of literals) constraints.

2.2 Compiling Models to Programs

Constraint modelling languages aim to provide high-level specification languages for constraint problems. Modern languages such as Zinc [Marriott *et al.*, 2008], Essence [Frisch *et al.*, 2008] and MiniZinc [Nethercote *et al.*, 2007] also aim to be *solver-independent*, i.e., they allow the user to specify the problem without committing to a particular solving technology. The challenge is to compile these solver-independent models into efficient solver-level programs.

Fig. 1 shows the typical compilation process, which is common to all modern modelling languages. In this paper we use the MiniZinc [Nethercote *et al.*, 2007] language and tool chain. Solver-level programs created by the MiniZinc compiler are expressed in a language called FlatZinc.

Instantiating a model's parameters with input data yields an *instance*. In order to solve a MiniZinc instance, it must be translated into the solver-dependent, flat representation of a FlatZinc program. This translation, called *flattening*, unrolls all loops and reduces expressions to sets of individual, flat constraints. The FlatZinc language is a subset of MiniZinc comprising only variable and constant declarations along with simple constraints. Each constraint in a FlatZinc program is a predicate call.

If one predicate or function call expression is nested as an argument of another, it is decomposed into a separate constraint, and the argument is replaced with an auxiliary variable holding the result of the call. For Boolean predicates that are not in a top-level conjunction, a reified version of the expression will be constructed and bound to an auxiliary Boolean variable.

The MiniZinc compiler includes predicate definitions from a constraint *library*. A predicate can be declared with a body of MiniZinc code that defines it in terms of simpler constraints, implementing it by *decomposition*. Alternatively, a predicate can be declared without a body marking it as a builtin. For each target solver, the MiniZinc system therefore contains a specialised library that defines all its builtins and provides solver specific decompositions for other constraints.

After translating all arguments of a call, the compiler looks for a predicate or function declaration with the same type signature. If it is defined as a builtin, the compiler inserts it into the FlatZinc as is. Otherwise, a copy of its body is instantiated with the actual arguments of the call and then flattened. Further optimisations performed by the MiniZinc compiler are detailed in Sect. 4.1.

2.3 Presolving

The concept of presolving was introduced by the mathematical programming community. Constructing a good MIP formulation can be quite difficult. Even small instances can require a large number of constraints and variables to be described correctly as a MIP program. These large programs can pose a difficulty for MIP solvers, due to either their combinatorial complexity, or simply because they require a large amount of memory to process. Presolving performs a whole program analysis that tightens the formulation of a problem. The regular, simple constraints in a MIP program allow for highly specialised and efficient analysis, with techniques ranging from simple bounds propagation, which tightens bounds on variables and constraints, to more advanced techniques such as probing.

The situation is similar for modern SAT solvers, which also try to improve the program by inferring information about the variables and by simplifying the constraints. Several presolving approaches also exist for CP solvers. Sect. 6 discusses these approaches in more detail.

3 Multi-Pass Presolving

This section presents our main contribution, an approach for communicating the results between different passes.

Traditional presolving as discussed above works at the *program* level, where all variables and constraints are known to the solver. Compared to this, a compiler for high-level modelling languages faces a challenge: it needs to ***make decisions during flattening*** that depend on its current knowledge of variable domains. For example, it can simplify a quadratic constraint `x*y` into a linear constraint if the value of `x` is known at compile time; it can turn a reified constraint `b <-> c` into a non-reified constraint `c` if `b` is known to be true; or it can avoid generating unnecessary code in loops such as `forall (i in lb(x)..ub(x)) (c(i))` if it can shrink the bounds of `x`.

More precise knowledge about variable domains can thus lead to shorter, more efficient FlatZinc, and potentially faster compilation and solving. However, when the compiler needs to make each individual decision, it has only partial knowledge about the whole program, because it has not yet compiled or analysed the entire model.

The solution is to compile at least twice: the initial passes collect information about the whole program, and subsequent passes use this information to produce better programs. The main novelty of our approach is that the different passes *do not need to compile for the same target*. As an example, the first pass may produce a program for a hypothetical ideal solver where all global constraints are built-ins. This pass would be quick, but could still reveal additional information. Alternatively, the first pass could target a presolver based on constraint propagation to infer new bounds. The second pass could then target a completely different solving technology, e.g. translating the problem for a MIP or SAT solver.

The design of MiniZinc, with its solver-specific libraries of constraint definitions, is ideal for this approach. Different passes can use different libraries. Multi-pass, multi-library presolving however poses a significant technical challenge: the different passes create different programs, with different variables, domains, and constraint decompositions. The compiler therefore needs to identify corresponding variables across passes, and communicate the information gathered about those variables from one pass to the next.

3.1 Multi-Pass Examples

The following examples show how additional information from multi-pass presolving can lead to better FlatZinc.

Listings 1–5 demonstrate the benefits of stronger presolving for models containing reified constraints. The model has three variables `x`, `y`, and `z` with domains $\{2,4\}$, $\{2,4\}$ and $\{2,4,5\}$ respectively. The first constraint is a typical `all_different` constraint while the second constraint introduces an implication between two MiniZinc expressions. Listing 5 shows the ideal FlatZinc that we would like the compiler to produce for this model.

Listing 2 is a simplified form of the FlatZinc resulting from compiling this model without any presolving. The first constraint is added directly to the FlatZinc as is. The implication is transformed to the disjunction $\neg($ `x+y+z=12` $) \vee$ `y = max([x,y,z])`. The negation is pushed inside the linear expression and Boolean control variables are introduced for both sides of the disjunction.

Listing 3 shows what happens when the compiler learns bounds for the top-level variable `z` by propagating the `all_different` constraint. The resulting FlatZinc is not very different. The variable `z` has been removed from some constraints since it is fixed but the resulting FlatZinc still contains two reified constraints and a redundant global constraint.

By running constraint propagation on the program given in Listing 2 the compiler can get bounds for all of the introduced variables. Listing 4 demonstrates what can be achieved by taking these new bounds into account while flattening. The

```
var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
constraint x+y+z=12 -> y=max([x,y,z]);
```

Listing 1: Original MiniZinc

```
var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
var 2..5: i0 = max([x,y,z])
var bool: b0 = (y = i0)
var bool: b1 = (x+y+z != 12)
constraint or(b0,b1);
```

Listing 2: Standard FlatZinc

```
var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
var 2..5: i0 = max([x,y,5])
var bool: b0 = (y = i0)
var bool: b1 = (x+y != 7)
constraint or(b0,b1);
```

Listing 3: Tightened top-level bounds

```
var {2,4}: x;  var {2,4}: y;  var {5}: z;
constraint x != y;
var {5}: i0 = max([x,y,5]) % remove (unused)
var {false}: b0 = (y = 5)  % remove (entailed)
var {true}: b1 = (x+y != 7)% drop reification
constraint or(b0,b1);      % remove (entailed)
```

Listing 4: Tightened intermediate bounds

```
var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
constraint x+y != 7;
```

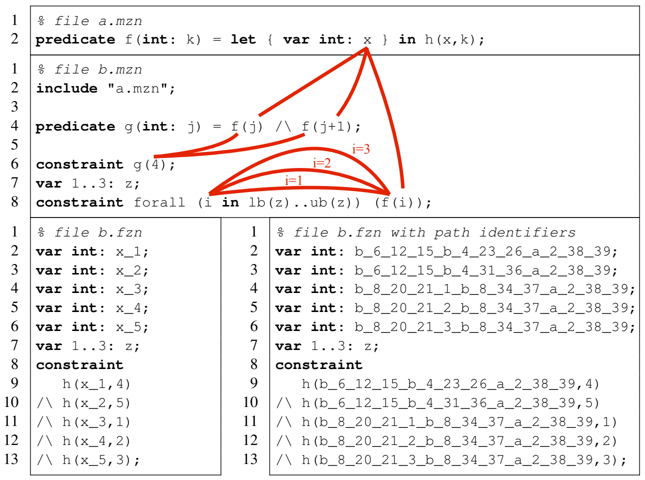
Listing 5: Desired FlatZinc

assignment to `i0` allows the compiler to remove the `max` constraint. With a value of 5 for `i0` the value of `b0` is trivially false leaving the disjunction with only one non-false disjunct. This forces the control variable `b1` to be true, allowing a non-reified version of the linear constraint to be used in place of the disjunction over the reified variable. With this additional information, the compiler will produce the desired FlatZinc shown in Listing 5. Note that the constraint `x+y != 7` is logically redundant, but the compiler only employs bounds reasoning and therefore cannot detect this currently.

Compiling this model for a MIP solver, presolving would result in a roughly 60% smaller FlatZinc program due to the complexity introduced by linearising the reified constraints.

3.2 Variable Synchronisation

In the example above, the set of variables generated in the first pass is the same as the set generated in the second pass. If this were guaranteed in general the compiler could update the domains of variables simply by matching variable names. In practise, bounds can be easily communicated for top-level variables that are common between different FlatZinc programs. However, the compiler cannot rely on the temporary variables introduced during flattening having the same names



```
1  % file a.mzn
2  predicate f(int: k) = let { var int: x } in h(x,k);
```

```
1  % file b.mzn
2  include "a.mzn";
3
4  predicate g(int: j) = f(j) /\ f(j+1);
5
6  constraint g(4);
7  var 1..3: z;
8  constraint forall (i in lb(z)..ub(z)) (f(i));
```

```
1  % file b.fzn
2  var int: x_1;
3  var int: x_2;
4  var int: x_3;
5  var int: x_4;
6  var int: x_5;
7  var 1..3: z;
8  constraint
9     h(x_1,4)
10 /\ h(x_2,5)
11 /\ h(x_3,1)
12 /\ h(x_4,2)
13 /\ h(x_5,3);
```

```
1  % file b.fzn with path identifiers
2  var int: b_6_12_15_b_4_23_26_a_2_38_39;
3  var int: b_6_12_15_b_4_31_36_a_2_38_39;
4  var int: b_8_20_21_1_b_8_34_37_a_2_38_39;
5  var int: b_8_20_21_2_b_8_34_37_a_2_38_39;
6  var int: b_8_20_21_3_b_8_34_37_a_2_38_39;
7  var 1..3: z;
8  constraint
9     h(b_6_12_15_b_4_23_26_a_2_38_39,4)
10 /\ h(b_6_12_15_b_4_31_36_a_2_38_39,5)
11 /\ h(b_8_20_21_1_b_8_34_37_a_2_38_39,1)
12 /\ h(b_8_20_21_2_b_8_34_37_a_2_38_39,2)
13 /\ h(b_8_20_21_3_b_8_34_37_a_2_38_39,3);
```

Figure 2: Variable paths

across compilations. The code in Fig. 2 exemplifies the problem of identifying variables across compilation passes.

The first MiniZinc file, `a.mzn`, defines a predicate `f` that introduces an integer variable `x`. The second file, `b.mzn`, includes `a.mzn` and defines a predicate `g` that calls `f` twice, as well as two constraints that call `g` and `f`, respectively.

Flattening will introduce five different instances of the variable `x`. The MiniZinc compiler will simply number them in the order they are generated. The first two, `x_1` and `x_2`, come from the call to `g` in line 6, and the remaining three are generated while iterating over the integers in the current bounds `1..3` of the variable `z`, in line 8. The resulting FlatZinc file `b.fzn` appears in the bottom left of Fig. 2.

Let us now have a look at the variable names generated in different compilation passes. Assume that the first pass narrowed the domain of `z` to the set `2..3`. In the second pass, iteration now starts at `i=2`, and only four instances of `x` are generated. Crucially, the names of the variables will change, too: the name `x_3` will now refer to the variable from iteration `i=2`, whereas in the first pass it referred to iteration `i=1`. Of course it would be wrong to transfer any information inferred about `x_3` during first pass presolving to the new second-pass variable `x_3`. Instead, information about first-pass `x_4` now corresponds to second-pass `x_3`.

3.3 Variable Paths

The example above makes it obvious that a multi-pass, multi-target presolving algorithm needs to *identify* variables that arise from the same sub-expressions *across passes*.

The algorithmic solution to this problem is to generate variable identifiers that are unique and that do not depend on the order of flattening. To satisfy these conditions it is sufficient that identifiers capture (1) the **call stack** leading to the introduction of the variable. In the example, `x_2` was generated from the call to `g` on line 6, and further the call `f(j+1)` on line 4. (2) Any **loop iteration values**. For `x_4`, the identifier was generated when `i` had the value 2.

We call an identifier that captures this information a **variable path**. The paths are represented graphically in Fig. 2, with lines connecting each predicate call through to the location where the variable is introduced.

Path representation. A textual representation can use the location in the source code to represent call sites and variable declarations, together with the value of loop variables to identify the iteration. For example, a path for `x_2` would include the location of the call `g(4)` in file `b.mzn` on line 6, column 12–15, the location of the call `f(j+1)` on line 4, column 31–36, and the location of `x` in `a.mzn`, line 2, column 38–39, resulting in the path `b:6.12-15,b:4.31-36,a:2.38-39`. For `x_4` we need to add the loop iteration value: `b:8.20-21,i=2,b:8.34-37,a:2.38-39`. The bottom right of Fig. 2 shows the paths (converted into valid MiniZinc identifiers).

Note how the identifiers are independent of the order in which the constraints are flattened or the concrete bounds of the variables at the time of flattening. For instance, the variable in line 5 now refers to the iteration `i=2`, independent of whether the loop started at 1 or at 2.

3.4 Presolving Phases

During and after a first compilation pass, the compiler can use several types of inference to compute stronger variable domains and discover variable aliases, which can then be communicated to the following pass using the path-based scheme introduced above.

Simplifications during flattening. Compilers for constraint modelling languages typically perform certain optimisations during flattening in order to produce tighter solver-level programs. In particular, compilers will perform *constant propagation*, *alias analysis*, and various methods of *aggregation and simplification* for particular constraints.

Post-flattening code optimisation. After completing the flattening, the compiler will again perform constant propagation, unify variables that have been found to be equal, and remove variables and constraints that are not used. For instance, a variable may have been introduced but all constraints that referenced it have been found to be entailed. Or a reified constraint may appear in a disjunction for which one disjunct has already been shown to be always true.

Constraint propagation. In addition to the optimisations performed during and after flattening, the compiler can run the propagation engine of a generic constraint solver, whose specialised propagation algorithms for various global constraints can produce tighter variable bounds or even assign variables. Furthermore, the solver can detect when constraints are entailed, leading to their removal from the model. One example of such an integration is the Savile Row compiler for Essence' [Nightingale *et al.*, 2014], which uses the Minion constraint solver to perform strong bounds propagation, resulting in more common subexpressions being detected and hence a better flat program. However, this approach only deals with top-level variables and cannot record bounds for introduced auxiliary variables.

4 Implementation in MiniZinc

This section discusses the implementation aspects and challenges of integrating multi-pass presolving into the MiniZinc 2.0 compiler [MiniZinc Team, 2015].

4.1 Compilation in MiniZinc

Compiling partial functions. For each sub-expression e, the compiler conceptually introduces two auxiliary variables, v_e for the value of e, and a Boolean b_e representing whether e is *defined* [Stuckey and Tack, 2013]. Consider e.g. the expression $e =$`x + (y div z) <= 10`. The relational semantics [Frisch and Stuckey, 2009] dictate that this constraint holds if and only if the division is defined (i.e. `z!=0`) and the condition is satisfied. *Advantage:* If b_e is true, later passes can compile the constraint as a total function.

Common Sub-expression Elimination (CSE). The compiler records every sub-expression in a map, so that when an identical expression is encountered again, the original result can be re-used. CSE has been described in the context of Essence [Rendl, 2010; Nightingale *et al.*, 2014]. *Advantage:* If two variables are fixed to the same value or found to be equal to each other, more sub-expressions can become identical and shared. Since the detection of common sub-expressions depends on the order of compilation, a single pass compiler may miss this.

Overloading Resolution. An important feature of MiniZinc is its support for overloading functions and predicates based on the types of their arguments. For example, an array access `x[y]` is compiled as an *element* constraint if `y` is a variable, but evaluated if fixed. When translating a function or predicate call, the compiler uses the types of the *actual* arguments to determine which version to use. *Advantage:* Fixing variables results in more specialised functions being applicable.

Linear Simplification. The linear parts of arithmetic expressions are aggregated into linear expressions. *Advantage:* Fixing variables and tightening their bounds will result in shorter linear constraints.

Boolean Simplification. Any complex Boolean expression needs to be decomposed into simple constraints understood by the target solver, such as conjunctions, disjunctions, clauses, xor, or negations. Before decomposing, the compiler normalises Boolean expressions by pushing negations inwards, and aggregates individual Boolean expressions into longer clauses and conjunctions. The compiler delays the decomposition of reified constraints as long as possible to avoid reification altogether if it can be inferred that the constraint must be globally true. *Advantage:* Fixing Boolean variables enables simplification of Boolean expressions, fewer disjunctions, and constraints being pushed into the top-level conjunction instead of being reified.

4.2 Implementing Variable Paths

Sect. 3.3 presented variable paths as identifiers, capturing the call stack and loop variables that lead to the variable being introduced. When generating large programs, e.g. when linearising for MIP solvers, often tens of thousands of variables are introduced. An actual implementation must therefore make both the generation and lookup of paths very efficient.

Instead of encoding the path into the variable identifier, the implementation keeps a separate *path map* from paths to variables. The initial compilation passes populate the map and each pass can check if the path of a newly introduced variable already exists in the map, in which case it can access the previous information.

The *call stack* is represented explicitly inside the compiler. It includes information about the recursive structure of the expression that is being compiled, including all comprehensions that introduce loop variables, in order to generate error messages. We re-use this data to construct variable paths, which are represented as strings stored in a simple hash map.

Optimised path generation. Since constructing and comparing paths as strings can be costly, the implementation employs two simple optimisations to reduce the number of paths generated. Firstly, we remember the maximum call stack depth from previous passes. Any time a variable is introduced deeper in the call stack, we know that it cannot be present in the map. Similarly, if a variable is introduced from a file that was not used in earlier passes, we also know that no presolving information can exist about it. In both cases, we save the overhead of constructing the path and looking it up in the map. These optimisations are critical for large models.

4.3 Presolving by Constraint Propagation

In addition to the simplifications performed by the MiniZinc compiler (as discussed in Sect. 3.4), we invoke the propagation engine of the Gecode [Gecode Team, 2006] constraint solver in the first compilation pass, taking advantage of Gecode's dedicated propagation algorithms for most of the MiniZinc global constraints to find tighter domains that are stored for use in further compilation passes.

5 Experiments

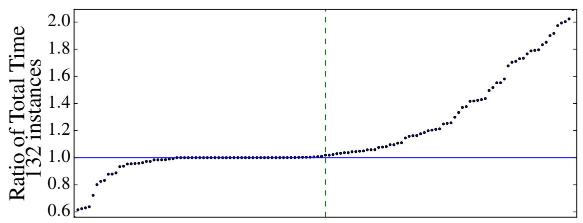
The experiments summarised in this section measure the impact of the new approach on compile-time overhead, program size and solving performance. We present results of two pass compilation where Gecode is used in the first pass. The experiments used machines with dual 2.00GHz Intel Quad Core Xeon E5405 processors with 16GB of RAM.

300 instances from the previous three years of MiniZinc Challenge problems were selected, covering 49 MiniZinc models. Each instance was compiled using the MiniZinc 2.0 compiler both with and without presolving. Compilation and solving both had an 8Gb memory limit.

Fig. 3 and 4 show summaries of the experimental results for two experiments, compiling for CP and MIP respectively. The CP solver in the experiments is Chuffed [Chu, 2011], a lazy clause generation solver that has often performed well in the MiniZinc challenge. We used CPLEX version 12.6 for the MIP experiments.

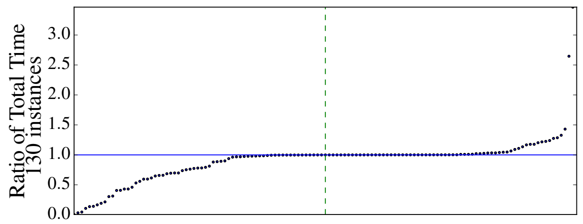
The tables show median (Med%) and geometric mean (Geo%) of the relative sizes and runtimes of presolved versus non-presolved programs. We report three subsets of instances for each experiment: the set of instances that compile without exceeding the memory limit; the set where presolving has changed the number of variables or constraints; and finally the set where solve times for both presolved and non-presolved programs are above one second and less than the time limit of 1800 seconds and so, can be compared.

Column *N* shows the number of instances in each group. Columns *Var*, *Con*, and *Dm* show the size of presolved programs relative to the non-presolved ones, measured in number of variables, constraints, and the product of domain sizes.



	N	*Var*	*Con*	*Dm*	*Cmp*	*Sol*	*Tot*
Med%	271	98	100	96	188	-	-
	132	90	92	91	194	100	101
	57	89	95	89	195	98	103
Geo%	271	90	92	85	170	-	-
	132	84	88	76	179	97	112
	57	85	89	75	187	93	105

Figure 3: CP Results (using Chuffed)



	N	*Var*	*Con*	*Dm*	*Cmp*	*Pre*	*Sol*	*Tot*
Med%	250	100	100	98	102	98	-	-
	130	97	97	91	96	90	100	100
	67	97	96	86	91	87	74	79
Geo%	250	96	97	89	106	86	-	-
	130	92	93	88	97	85	72	78
	67	88	90	85	96	80	53	61

Figure 4: MIP Results (using CPLEX)

The columns *Cmp*, *Pre*, *Sol*, and *Tot* denote the duration of compilation, CPLEX's own presolve, solving and the total compile and solve time. The accompanying plots show the sorted individual ratios of total time for changed instances. The plateau mostly comprises timeouts. The vertical dashed line indicates the median.

Compiling Constraint Programs For Chuffed, some reduction in program size and solving time was expected. A large increase in relative compilation time is also expected, since each instance is flattened twice with similar libraries.

Fig. 3 shows that on the full set of 271 instances we often have a reduction in problem size with a typical compilation time increase of 70%. The compile time increases with the impact of presolving, with an increase of 87% in the group of comparable instances. Solve time is often reduced but does not pay for the compilation time in many cases.

Further analysis of the results suggests that we can identify two classes of problems where presolving hurts performance. First, when the total time is dominated by compilation time. In this case, compiling twice adds a significant overhead. We can address this issue by limiting the time available for presolving. In the second case, solving the presolved model takes much longer than the non-presolved. Here the presolving appears to have some negative effect on the search strategy and the nogoods learned by during search. We will have to further investigate how to detect these cases.

Compiling Mixed-Integer Linear Programs Linearising a constraint model for a MIP solver often requires decompositions that can introduce many variables. Compared to the CP experiments above, we therefore expect the strength of multi-pass presolving to be much more pronounced.

To control for the erratic behaviour of heuristics in MIP solvers [Fischetti and Monaci, 2014] we ran each instance six times with different random seeds. To narrow the set of 250 instances we looked at the results of running CPLEX's presolver on each program and kept the 130 instances where multi-pass presolving resulted in a change to the number of variables or constraints *after* MIP presolving.

As expected we see a much larger impact of multi-pass compilation when applied to MIP, where total time improves by 39% (geometric mean) in the 67 comparable instances. This improvement is due to the solve time being almost halved on average. Due to how comparatively cheap it is to compile for CP first the presolving has a small cost of around 6% for the whole set of instances, and actually speeds up compilation in many cases, most likely due to a reduction in the amount of decomposition required. Detailed examination of the results suggested that the biggest improvements are gained where presolving can reduce the number of element and reified linear constraints. The results show that our presolving technique is very effective for a MIP target.

6 Related Work

Presolving originated in the field of Linear Programming, and has been a core part of any LP solver for decades. Most of these methods aim at tightening variable bounds and removing redundant constraints. For a survey of presolving methods in LP see [Andersen and Andersen, 1995].

A number of dedicated presolving algorithms are in use that target the discrete variables in a MIP model [Mahajan, 2010]. An important technique is *probing* [Savelsbergh, 1994], which tentatively assigns 0/1 variables and observes the effects of the assignment.

SAT solvers face a similar challenge as LP and MIP solvers, in that their input programs can be huge. This has led to the development of presolvingechniques for SAT [Eén and Biere, 2005], which also aim at fixing variables and removing redundant clauses. *Equi-propagation* [Metodi *et al.*, 2013] is a high-level approach that starts with a model that includes integer variables and constraints, resulting in SAT encodings that are much more efficient than the direct encoding without integer-level propagation. For problems that have a large Boolean component, these techniques may be interesting as a multi-pass presolving step.

The use of constraint programming to preprocess combinatorial optimisation problems before solving using another technology has been explored in the past [Hooker, 2006; Achterberg, 2009]. The *Savile Row* compiler for Essence' performs multi-pass presolving using *Minion* as a constraint propagation engine [Nightingale *et al.*, 2014]. The Essence' language does not support user-defined predicates and functions, and solver-specific translation routines are hard-coded into the compiler. To the best of our knowledge, the compiler can only communicate presolving results through top-level model variables. In contrast, our approach is compatible with the high-level problem structure as expressed in predicate and function definitions and calls, by enabling presolving on introduced auxiliary variables.

7 Conclusion and Future Work

We introduced an improved compilation framework that integrates whole program optimisation with a novel variable synchronisation method, allowing tighter domains and variable aliases to be communicated between several distinct representations of the same instance. Further, we presented evaluations of the approach when applied to two different targets. For a CP solver, we found that in many cases compilation is too slow to justify using multi-pass presolving in all cases. With a MIP target we see very little compilation overhead and a reduction in program size and solving times. In conclusion, multi-pass presolving can make the compilation of constraint models more robust, with a low average time overhead, and the potential to improve performance significantly.

There are several applications of the new variable synchronisation method that we would like to explore in future. One example is the automatic construction of hybrid solvers that communicate on more than just top-level variables. It may also be interesting to explore the benefits of presolving iteratively until a fixed-point is reached and of synchronising additional information such as constraint entailment. We also hope to use other inference approaches, for instance supporting arbitrary precision integers and infinities.

Acknowledgements. We thank Peter Nightingale and Christopher Jefferson for open discussions about *Savile Row*, and Maria Garcia de la Banda for helping prepare this paper. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This research was partly sponsored by the Australian Research Council grant DP140100058.

References

[Achterberg, 2009] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009.

[Andersen and Andersen, 1995] Erling D. Andersen and Knud D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71(2):221–245, 1995.

[Chu, 2011] Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, The University of Melbourne, 2011.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[Fischetti and Monaci, 2014] Matteo Fischetti and Michele Monaci. Exploiting erraticism in search. *Operations Research*, 62:114–122, 2014.

[Frisch and Stuckey, 2009] A. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In I. Gent, editor, *CP*, volume 5732 of *LNCS*, pages 367–382. Springer, 2009.

[Frisch *et al.*, 2008] Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[Gecode Team, 2006] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[Hooker, 2006] John N. Hooker. *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer, 2006.

[Mahajan, 2010] A. Mahajan. Presolving mixed-integer linear programs. Technical report, Argonne National Laboratory, 2010.

[Marriott *et al.*, 2008] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[Metodi *et al.*, 2013] Amit Metodi, Michael Codish, and Peter J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013.

[MiniZinc Team, 2015] MiniZinc Team. MiniZinc 2.0, 2015. Available from `http://www.minizinc.org`.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Nightingale *et al.*, 2014] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In Barry O'Sullivan, editor, *CP*, volume 8656 of *LNCS*, pages 590–605. Springer, 2014.

[Rendl, 2010] Andrea Rendl. *Effective Compilation of Constraint Models*. PhD thesis, School of Computer Science, University of St Andrews, 2010.

[Savelsbergh, 1994] Martin W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *INFORMS Journal on Computing*, 6(4):445–454, 1994.

[Stuckey and Tack, 2013] Peter J. Stuckey and Guido Tack. MiniZinc with functions. In Carla Gomes and Meinolf Sellmann, editors, *CPAIOR*, volume 7874 of *LNCS*, pages 268–283. Springer, 2013.