Automated Auxiliary Variable Elimination through On-the-Fly Propagator Generation

Jean-Noël Monette, Pierre Flener, and Justin Pearson

Uppsala University, Department of Information Technology, Uppsala, Sweden
{jean-noel.monette,pierre.flener,justin.pearson}@it.uu.se

Abstract. Model flattening often introduces many auxiliary variables. We provide a way to eliminate some of the auxiliary variables occurring in exactly two constraints by replacing those two constraints by a new equivalent constraint for which a propagator is automatically generated on the fly. Experiments show that, despite the overhead of the preprocessing and of using machine-generated propagators, eliminating auxiliary variables often reduces the solving time.

1 Introduction

Constraint-based modelling languages such as Essence [6] and MiniZinc [12] enable the modelling of problems at a higher level of abstraction than is supported by most constraint solvers. The transformation from a high-level model to a low-level model supported by a solver is often called *flattening* and has been the subject of intense research in order to produce good low-level, or *flattened*, models (see, e.g., [14]). By decomposing complex expressions into a form accepted by a solver, flattening often introduces many auxiliary variables into the flattened model. Those auxiliary variables and the propagators of the constraints in which they appear may have a large negative impact on the efficiency of solving (as we will show in Section 5).

In this paper, we propose a fully automated way to address this problem by removing from the flattened model some auxiliary variables that appear in exactly two constraints and replacing those two constraints by a new equivalent constraint for which a propagator is generated. Given a flattened model, our approach is fully automated and online. It can be summarised as follows:

1. Identify frequent patterns in the flattened model consisting of two constraints sharing an auxiliary variable.
2. For a pattern, define a new constraint predicate that involves all variables appearing in the pattern except for the shared auxiliary variable.
3. For a pattern, replace all its occurrences in the flattened model by instantiating the new constraint predicate.
4. For each new predicate, generate a propagator description in the indexical language of [11] and compile it for the targeted constraint solver.
5. Solve the modified flattened model using the constraint solver extended with the new indexical-based propagators.

Our experiments in Section 5 show that our approach is useful for instances that are hard to solve, reducing the average time by 9% for those taking more than one minute and sometimes more than doubling the search speed.

The rest of the paper assumes that the low-level language is FlatZinc [1] and that the constraint solver is Gecode [7], as our current implementation is based on FlatZinc and Gecode. However, the ideas presented here are applicable to other low-level modelling languages and other constraint solvers.

The paper is organised as follows. We start with some preliminaries in Section 2. Then we describe our approach in Section 3 and present a complete example in Section 4. In Section 5, we show that this approach is effective in practice. In Section 6, we discuss the merits and limitations of our approach as well as some alternatives. Finally, we conclude in Section 7.

2 Preliminaries

A *constraint-based model* is a formal way to describe a combinatorial problem. It is composed of a set of variables, each with a finite domain in which it must take its value, a set of constraints, and an objective. A *solution* is an assignment to all variables so that all constraints are satisfied. The *objective* is either to find a solution, or to find a solution in which a given variable is minimised, or to find a solution in which a given variable is maximised.

A *constraint predicate*, or simply *predicate*, is defined by a name and a *signature* that lists the *formal arguments* of the predicate. The arguments can be constants, variables, or arrays thereof. For simplicity, we identify a predicate with its name and we often do not give the types of its arguments. We specify the *semantics* of a predicate P by a logic formula involving the arguments of P and constants. For example, one could write $\text{PLUS}(X, Y, Z) \triangleq X = Y + Z$, for the predicate PLUS with formal arguments X, Y, and Z. A predicate whose signature has arrays of variables is said to be of unfixed arity, or n-ary. Unfixed-arity predicates are usually referred to as global constraints [2] in constraint programming. Each constraint in a constraint-based model is the *instantiation* of a constraint predicate to some *actual arguments*. We denote formal arguments in upper case and actual arguments inside a model in lower case.

2.1 MiniZinc and FlatZinc

MiniZinc is a solver-independent constraint-based modelling language for combinatorial problems. Before being presented to a solver, a MiniZinc model is transformed into a FlatZinc model by a process called *flattening*. The MiniZinc flattener inlines function and predicate calls, decomposes expressions, and unrolls loops to provide the solver with a model that is a conjunction of primitive constraints (i.e., constraints that are recognised by the targeted solver) over simple arguments (i.e., only variables, constants, or arrays thereof). To do so, the flattener may introduce auxiliary variables, which are annotated in the FlatZinc model with `is_introduced_var`.

```
1 var int: w;
2 var int: y;
3 var int: z;
4
5
6
7 constraint w≠max(y,z) ∨ 1≤z;
8
9
10
11 solve satisfy;
```

(a) MiniZinc model

```
1 var int: w;
2 var int: y;
3 var int: z;
4 var int: x_1 ::var_is_introduced;
5 var bool: x_2 ::var_is_introduced;
6 var bool: x_3 ::var_is_introduced;
7 constraint x_1 = max(y,z);
8 constraint x_2 ≡ w ≠ x_1;
9 constraint x_3 ≡ 1 ≤ z;
10 constraint x_2 ∨ x_3;
11 solve satisfy;
```

(b) FlatZinc model

Fig. 1: MiniZinc model (1a) and FlatZinc model resulting from its flattening (1b)

For example, the MiniZinc model of Figure 1a is flattened into the FlatZinc model of Figure 1b, modulo editorial changes. The single constraint expression of the MiniZinc model (line 7 in Figure 1a) is flattened by introducing three auxiliary variables (lines 4–6 in Figure 1b) and posting four primitive constraints: the constraints in lines 7–9 functionally define the auxiliary variables representing parts of the original constraint expression, and the disjunction of line 10 corresponds to the top-level expression.

For 75 of the 488 benchmark instances we used (see Section 5 for details), no auxiliary variables are introduced by flattening. For 264 instances, flattening multiplies the number of variables in the model by more than 5. For 7 instances, flattening even multiplies the number of variables by more than 100.

2.2 Patterns, Occurrences, and Extensions

A *pattern* is here a new constraint predicate with signature $Y_1, ..., Y_n, K_1, ..., K_m$, where the Y_i are $n \geq 2$ variable identifiers and the K_i are $m \geq 0$ constant identifiers. Its semantics is specified by the conjunction of two existing predicates P_A and P_B applied to arguments $A_1, \ldots, A_p$ and $B_1, \ldots, B_q$, for some $p, q \geq 2$, such that each A_i and each B_i is either one of $Y_1, \ldots, Y_n, K_1, \ldots, K_m$, or a unique local and existentially quantified variable X, or a constant, and such that X appears in both $A_1, \ldots, A_p$ and $B_1, \ldots, B_q$. We reserve the identifier X for the local and existentially quantified variable. Hence, for simplicity, we will often omit writing the existential quantification of X in predicate semantics.

An *occurrence* of a pattern in a model is made of two constraints C_1 and C_2 sharing a variable x that appears only in C_1 and C_2 such that P_A can be instantiated to C_1 and P_B to C_2 with X being instantiated to x.

For example, the pattern $P(Y_1, Y_2, Y_3, Y_4) \triangleq \exists X : X = \max(Y_1, Y_2) \wedge Y_3 \equiv Y_4 \neq X$ occurs in the model of Figure 1b, with C_1 being $x_1 = \max(y, z)$ in line 7, C_2 being $x_2 \equiv w \neq x_1$ in line 8, and X being x_1. Hence this occurrence of P is equivalent to the constraint $P(y, z, x_2, w)$.

```
1  def PLUS(vint X, vint Y, vint Z){
2    propagator{
3      X in (min(Y)+min(Z)) .. (max(Y)+max(Z)) ;
4      Y in (min(X)-max(Z)) .. (max(X)-min(Z)) ;
5      Z in (min(X)-max(Y)) .. (max(X)-min(Y)) ;
6    }
7    checker{ X == Y + Z }
8  }
```

Fig. 2: Indexical propagator description for $X = Y + Z$

A pattern P_1 is said to be a *specialisation* of another pattern P_2 if one can define P_1 in terms of P_2 by properly instantiating some of the arguments of P_2.

For example, consider the patterns $P_1(Y_1, Y_2) \triangleq (X \vee Y_1) \ \wedge \ X \equiv 1 \leq Y_2$ and $P_2(Y_1, Y_2, K_1) \triangleq (X \vee Y_1) \ \wedge \ X \equiv K_1 \leq Y_2$, both of which occur on lines 9 and 10 of Figure 1b with X being x_3. Then P_1 is a specialisation of P_2 because $P_1(Y_1, Y_2) \Leftrightarrow P_2(Y_1, Y_2, 1)$.

2.3 Indexicals

Indexicals [20] are used to describe concisely propagation in propagator-based constraint solvers. The core indexical expression takes the form 'X `in` σ' and restricts the domain of variable X to be a subset of the set-valued expression σ, which depends on the domains of other variables. Figure 2 presents an indexical propagator description in the language of [11] for the constraint predicate $\text{PLUS}(X, Y, Z) \triangleq X = Y + Z$; it involves three indexical expressions, one for each variable (lines 3–5), the `min` and `max` operators referring to the domain of the argument variable. Figure 2 also contains a *checker* (line 7), which is used to test whether the constraint holds on ground instances and can be seen as a specification of the predicate semantics.

Our indexical compiler [11] takes an indexical propagator description like the one in Figure 2 and compiles it into an executable propagator for a number of constraint solvers, including Gecode. The compiled propagator is said to be *indexical-based*. The experimental results in [11] show that an indexical-based propagator uses between 1.2 and 2.7 times the time spent by a hand-crafted propagator on a selection of n-ary constraint predicates.

3 Our Approach

Given a FlatZinc model, our approach, whose implementation is referred to as `var-elim-idxs`, adds a preprocessing step before solving the model. This preprocessing is summarised in Algorithm 1. We do not describe lines 1, 5, and 6, as they involve purely mechanical and well-understood aspects. The core of the algorithm iteratively identifies frequent patterns (line 2), replaces them in the

Algorithm 1 Main preprocessing algorithm of `var-elim-idxs`
Input: a flattened model
Output: updated flattened model and extended solver

1: Parse the flattened model
2: **while** there is a most frequent pattern P in the model (Section 3.1) **do**
3: Replace each occurrence of P in the model by instantiating P (Section 3.2)
4: Generate and compile an indexical propagator description for P (Section 3.3)
5: Output the updated model
6: Link the compiled indexical-based propagators with the solver

model (line 3), and generates propagators (line 4). Sections 3.1 to 3.3 describe this core.

The loop of lines 2 to 4 ends when no pattern occurs often enough, which is when the number of occurrences of a most frequent pattern (if any) is less than 10 or less than 5% of the number of variables in the model: under those thresholds, the effort of preprocessing is not worth it. To save further efforts, if the number of auxiliary variables is less than 10 or less than 5% of the number of variables in the model (this criterion can be checked very fast with Unix utilities such as `grep`), then the preprocessing is not performed at all. The two thresholds, 10 and 5%, have been set arbitrarily after some initial experiments.

3.1 Identification of Frequent Patterns

We now show how we identify patterns that occur frequently in a given model.

First, we collect all auxiliary variables that appear in exactly two constraints. Indeed, to be in an occurrence of a pattern, a shared variable must occur only in those two constraints. Most auxiliary variables appear in exactly two constraints: this is the case of 84% of the auxiliary variables appearing in the 488 benchmark instances we used. For example, this is also the case for all auxiliary variables in Figure 1b. However, due to common subexpression elimination [13, 14], some auxiliary variables appear in more than two constraints. We do not consider such variables for elimination as common subexpression elimination usually increases the amount of filtering and eliminating those variables might cancel this benefit.

Then, for each collected variable x, we create a pattern as follows. Let $P_A(a_1, \ldots, a_p)$ and $P_B(b_1, \ldots, b_q)$ be the two constraints in which x appears. The pattern is such that the two predicates are $P_A(A_1, \ldots, A_p)$ and $P_B(B_1, \ldots, B_q)$, where each A_i is defined by the following rules, and similarly for each B_i:

- If a_i is x, then A_i is X.
- If a_i is a variable other than x, then A_i is Y_k for the next unused k.
- If $a_i \in \{-1, 0, 1, \mathbf{true}, \mathbf{false}\}$, then A_i is a_i.
- If a_i is a constant not in $\{-1, 0, 1, \mathbf{true}, \mathbf{false}\}$, then A_i is K_k for the next unused k.
- If a_i is an array, then A_i is an array of the same length where each element is defined by applying the previous rules to the element of a_i at the same position.

The purpose of the third rule is to allow a better handling of some special constants, which may simplify the generated propagators. For example, linear (in)equalities can be propagated more efficiently with unit coefficients.

In general, there might be other shared variables between $P_A(a_1, \ldots, a_p)$ and $P_B(b_1, \ldots, b_q)$ besides x but, to keep things simple, we consider them separately, i.e., a new argument Y_k with an unused k is created for *each* occurrence of another shared variable. Ignoring other shared variables does not affect the correctness of our approach, but the generated propagators may achieve less filtering than possible otherwise. We will revisit this issue in Section 6.

In order to avoid the creation of symmetric patterns, we sort the elements of an array when their order is not relevant: we do this currently only for the n-ary Boolean disjunction and conjunction constraints.

For example, both $(x \vee y) \wedge (x \equiv 1 \leq z)$ and $(w \vee u) \wedge (u \equiv 1 \leq v)$ are considered occurrences of $P(Y_1, Y_2) \triangleq (X \vee Y_1) \wedge (X \equiv 1 \leq Y_2)$, although, syntactically, the shared variables (x and u, respectively) occur in different positions in their respective disjunctions.

Finally, we count the occurrences of each created pattern. In doing so, we ignore the following patterns, expressed here in terms of criteria that are specific to FlatZinc and Gecode:

- Patterns involving an n-ary predicate with at least five variables among its arguments: currently, our approach considers only fixed-arity patterns. Hence n-ary constraints with different numbers of variables are considered as fixed-arity constraints with different arities. We only want to keep those of small arities for efficiency reasons. The threshold is set to 5 because no fixed-arity predicate in FlatZinc has more than four variable arguments.
- Patterns involving a predicate for which an indexical-based propagator is expected to perform poorly with respect to a hand-written propagator: this includes all the global constraint predicates from the MiniZinc standard library, as well as the element, absolute value, division, modulo, and multiplication predicates of FlatZinc.
- Patterns involving a `bool2int` predicate in which the Boolean variable is the shared variable, and another predicate that is not a pattern itself: this case is ignored as, in some cases, the Gecode-FlatZinc parser takes advantage of `bool2int` constraints and we do not want to lose that optimisation.

The two first criteria partially but not completely overlap: for example, the constraint `all_different`$([X, Y, Z, W])$ is not ruled out by the first criterion as it has fewer than five variables, but it is ruled out by the second one; conversely, `int_lin_eq`$(\text{coeffs}, [X, Y, Z, W, U], k)$ is ruled out by the first criterion but not by the second one.

The result of pattern identification is a pattern that occurs most in the model.

3.2 Pattern Instantiation

Having identified a most frequent pattern P, we replace each occurrence of P by an instantiation of P. More precisely, for each auxiliary variable x that appears

in exactly two constraints, if the pattern created for x in Section 3.1 is P or a specialisation of P (detected by simple pattern matching), then we replace in the model the declaration of x and the two constraints in which x appears by an instantiation of P, obtained by replacing each formal argument of P by the actual arguments of the two constraints in which x appears. To achieve this, each argument A_i of P_A in the semantics of P is considered together with the argument a_i of the instantiation of P_A in which x appears, in order to apply the following rules, and similarly for each B_i:

- If A_i is X, then variable a_i, which is x, is not in the instantiation of P.
- If A_i is Y_k, then Y_k is instantiated to the variable a_i.
- If $A_i \in \{-1, 0, 1, \mathbf{true}, \mathbf{false}\}$, then a_i is not in the instantiation of P.
- If A_i is K_k, then K_k is instantiated to the constant a_i.
- If A_i is an array, then the previous rules are applied to each element of A_i.

For example, consider the pattern $P_1(Z_1, Z_2) \triangleq (X \equiv 1 \leq Z_1) \ \wedge \ (Z_2 \vee X)$. Variable x_3 of Figure 1b appears in $x_3 \equiv 1 \leq z$ (line 9) and $x_2 \vee x_3$ (line 10). Then P_1 can be instantiated to $P_1(z, x_2)$ and this constraint replaces lines 6, 9, and 10 in Figure 1b.

Due to the sorting of the elements for the n-ary Boolean conjunction and disjunction predicates, some occurrences of a pattern may disappear before their instantiation. Consider the MiniZinc-level expression $z_1 > 0 \vee z_2 > 0$. Flattening introduces two auxiliary Boolean variables, say b_1 and b_2, together with the three constraints $b_1 \equiv z_1 > 0$, $b_2 \equiv z_2 > 0$, and $b_1 \vee b_2$. Hence there are two occurrences of the pattern $P(Y_1, Y_2) \triangleq (X \equiv Y_1 > 0) \wedge (X \vee Y_2)$ but only one of them will actually be replaced, say the one in which X is b_1. After replacing it, the model contains the two constraints $P(z_1, b_2)$ and $b_2 \equiv z_2 > 0$, changing the pattern to which b_2 belongs. This new pattern might be replaced in a later iteration of the loop in Algorithm 1.

3.3 Indexical Propagator Description Generation and Compilation

The generation of a propagator for a pattern uses our indexical compiler [11] and performs the following steps:

1. Translation of the pattern into a checker in the indexical syntax.
2. Elimination of the shared variable from the checker.
3. Generation of an indexical propagator description, based on the checker.
4. Compilation [11] of the indexical propagator description into an actual propagator, written in C++ in the case of Gecode.

Step 1 only involves a change of syntax, and Step 4 has already been described in [11]. Hence, we will focus here on Steps 2 and 3.

Let X be the variable to eliminate in Step 2. Here, variable elimination can take two forms. First, if X is constrained to be equal to some expression ϕ in one of the two conjuncts, i.e., if the checker can be written as $P(\dots, X, \dots) \wedge X = \phi$ for some predicate P, then all occurrences of X in the checker are replaced by

ϕ, i.e., the checker becomes $P(\ldots, \phi, \ldots)$. As Boolean variables are considered a special case of integer variables in the indexical compiler, this rule also covers the case of X being a Boolean variable constrained to be equivalent to a Boolean expression. Second, if both conjuncts are disjunctions, one involving a Boolean variable X and the other $\neg X$, i.e., if the checker can be written as $(\delta_1 \vee X) \wedge (\delta_2 \vee \neg X)$ for some Boolean expressions δ_1 and δ_2, then applying the resolution rule yields a single disjunction without X, i.e., the checker becomes $\delta_1 \vee \delta_2$.

The generation in Step 3 of an indexical propagator description from a checker works by syntactic transformation: rewriting rules are recursively applied to the checker expression and its subexpressions in order to create progressively a collection of indexical expressions. The whole transformation has more than 250 rules. We limit our presentation to the most representative ones.

The rule for a conjunction $\gamma_1 \wedge \gamma_2$ concatenates the rewriting of γ_1 and γ_2. The rule for a disjunction $\delta_1 \vee \delta_2$ is such that δ_2 is propagated only once δ_1 can be shown to be unsatisfiable, and conversely. The rule for an equality $\phi_1 = \phi_2$ creates expressions that force the value of ϕ_1 to belong to the possible values for ϕ_2, and conversely. If ϕ_1 is a variable, say Y, then the rule creates the indexical Y `in UB(`ϕ_2`)`, where `UB(`ϕ_2`)` is possibly an over-approximation of the set containing all possible values of the expression ϕ_2, based on the domains of the variables appearing in ϕ_2. If ϕ_1 is not a variable but a compound expression, then it must be recursively decomposed. Consider the case of ϕ_1 being $Y_1 + Y_2$: two indexical expressions are created, namely Y_1 `in UB(`$\phi_2 - Y_2$`)` and Y_2 `in UB(`$\phi_2 - Y_1$`)`. The other rules cover all the other expressions that can appear in a checker. The function `UB(.)` is also defined by rules. As an example, `UB(`$\phi_1 - \phi_2$`)` is rewritten as `UB(`ϕ_1`)` $\ominus$ `UB(`ϕ_2`)`, where $\ominus$ is pointwise integer set substraction: $S \ominus T = \{s - t \mid s \in S \wedge t \in T\}$.

This generation mechanism packaged with our indexical compiler has been used in [9,17] to prototype propagators rapidly for newly identified constraints, but it has never been described before. It is very similar to the compilation of projection constraints in Nicolog [18], but generalised to n-ary constraints.

4 Example: Ship Schedule

To illustrate our approach, we now consider the `ship-schedule.cp.mzn` model from the MiniZinc Challenge 2012. The objective of the problem is to find which boats are sailing and when, in order to satisfy several port constraints, e.g., tides, tugboat availability, and berth availability, as well as to maximise the total weight that can be transported. The FlatZinc model produced by flattening the MiniZinc model with the `7ShipsMixed.dzn` data file, which represents an instance with 7 boats and 74 time slots, contains 7,187 constraints and 5,978 variables, among which 4,848 are auxiliary, i.e., 81% of the variables.

When given this flattened model, `var-elim-idxs` iteratively identifies the patterns reported in Table 1. Note that pattern P_0 is used in pattern P_1. The loop of Algorithm 1 ends upon identifying pattern P_4, which occurs less often than 5% of the number of variables and is not instantiated. In total, 4 new constraint

Table 1: Patterns found in `ship-schedule.cp.mzn` with `7ShipsMixed.dzn`

Predicate		Definition			Frequency
$P_0(Y_1, Y_2)$	$\triangleq$	$(X \vee Y_1)$	$\wedge$	$X \equiv (Y_2 = 0)$	892
$P_1(Y_1, Y_2, Y_3, Y_4)$	$\triangleq$	$P_0(X, Y_4)$	$\wedge$	$X \equiv (Y_1 \wedge Y_2 \wedge Y_3)$	612
$P_2(Y_1, Y_2, Y_3, Y_4, K_1)$	$\triangleq$	$(X \vee \neg(Y_1 \wedge Y_2 \wedge Y_3))$	$\wedge$	$X \equiv (Y_4 = K_1)$	612
$P_3(Y_1, Y_2, Y_3, K_1)$	$\triangleq$	$(X \vee \neg(Y_1 \wedge Y_2))$	$\wedge$	$X \equiv (Y_3 = K_1)$	276
$P_4(Y_1, Y_2, Y_3, Y_4)$	$\triangleq$	$(X \vee Y_1)$	$\wedge$	$X \equiv (Y_2 \wedge Y_3 \wedge Y_4)$	146

```
1  def P_1(vint Y_1::Bool, vint Y_2::Bool, vint Y_3, vint Y_4){
2    checker{
3      0==Y_4 or (1 <= Y_1 and 1 <= Y_2 and 1 <= Y_3)
4    }
5    propagator(gen)::DR{
6      once(not 0 memberof dom(Y_4)){
7        Y_1 in 1 .. sup;
8        Y_2 in 1 .. sup;
9        Y_3 in 1 .. sup;
10     }
11     (max(Y_1) < 1 or max(Y_2) < 1 or max(Y_3) < 1) -> Y_4 in {0};
12   }
13 }
```

Fig. 3: Indexical propagator description generated for pattern P_1 of Table 1 from `ship-schedule.cp.mzn` with `7ShipsMixed.dzn`. Note that the indexical compiler treats Boolean variables as a special case of integer variables with domain $\{0, 1\}$, where 1 represents **true**. For example, `1 <= Y_1` means `Y_1` is **true**.

predicates (P_0 to P_3) are introduced in the model, eliminating 2,392 variables, i.e., 40% of the variables. Among those, 222 variables are eliminated thanks to pattern specialisation, i.e., because the pattern created for such a variable is not exactly one of the patterns shown in Table 1 but a specialisation of one of those, for example a specialisation of P_2 with $K_1 = 1$.

The final model contains 3,586 variables, of which 2,456 are auxiliary, and 4,795 constraints, of which 1,780 use the generated propagators. Many auxiliary variables are not eliminated because either they appear in more than two constraints, or they appear in too infrequent patterns, such as P_4 in Table 1, or they appear in constraints that are not handled by `var-elim-idxs`, such as n-ary constraints with $n \geq 5$, the multiplication constraint, and the `bool2int` constraint in the case explained in Section 3.1.

The new constraint predicates are handled by the indexical compiler to specify their checkers, generate indexical propagator descriptions, and compile the latter into Gecode propagators. Figure 3 shows the generated indexical propagator description for pattern P_1 of Table 1. It is interesting to note that the auxiliary variable X eliminated by instantiating P_1 represents the truth of $Y_1 \wedge Y_2 \wedge Y_3$.

Hence the propagator for $X \equiv (Y_1 \wedge Y_2 \wedge Y_3)$ in the original flattened model needs to watch both when all conjuncts become **true** to set X to **true** and when some conjunct becomes **false** to set X to **false**. In contrast, the propagator of Figure 3 is only interested in the falsity of the conjuncts to restrict Y_4 to 0. The generated C++ code of the indexical-based propagators for Gecode is 579 lines long.

While the unmodified Gecode-FlatZinc interpreter solves the flattened model in 111.5 seconds, `var-elim-idxs` solves it in 89.1 seconds, divided into 4.9 seconds of preprocessing and 84.2 seconds of actual solving.

5 Experimental Evaluation

We implemented the preprocessing step of `var-elim-idxs`[1] in Scala, using our indexical compiler[2] and the FlatZinc parser of the OscaR project[3] as that parser is written in Scala as well. For the experiments, we used Gecode 4.4.0[4] and the MiniZinc flattener 2.0.1.[5] The experiments were carried out inside a VirtualBox virtual machine running Ubuntu 14.04 LTS 32-bit, with access to one core of a 64-bit Intel Core i7 at 3 GHz and 1 GB of RAM.

We tested `var-elim-idxs` on 489 of the 500 FlatZinc instances from the MiniZinc Challenges 2010 to 2014. We excluded 11 instances that could not be flattened, for lack of memory or because of a syntax error.[6] We ran both Gecode and `var-elim-idxs` once on each instance with a time-out of 10 minutes per instance. Unless otherwise noted, 'Gecode' refers here to running an unmodified version of Gecode and '`var-elim-idxs`' refers to running our preprocessing step followed by running the extended version of Gecode.

For 94 instances, the preprocessing is not run at all because the `grep` command detects that there are too few auxiliary variables: the behaviour of Gecode and `var-elim-idxs` is identical as the time spent by `grep` is negligible. For 172 instances, the preprocessing is run but does not identify any frequent enough pattern: the behaviour of Gecode and `var-elim-idxs` is identical except for the extra time spent on preprocessing, discussed in the last paragraph of this section.

Table 2 reports the results, aggregated per MiniZinc model, on the 223 instances in which the preprocessing identifies frequent patterns. We refer to those instances as *modified* instances. The bottom of the table presents the aggregated results over all modified instances for which the total time of Gecode is respectively more than 1 and 60 seconds. The node rate ratio for an instance is computed as r_v/r_g, where $r_x = n_x/t_x$ with n_x being the number of nodes of the search tree visited before time-out, and t_x being the total time in the column *ratio total* and the search time in the column *ratio search*; the subscript

[1] `https://bitbucket.org/jmonette/var-elim-idxs`
[2] `https://bitbucket.org/jmonette/indexicals`
[3] `http://www.oscarlib.org`
[4] `http://www.gecode.org/`
[5] `http://www.minizinc.org/2.0/`
[6] The *sugiyama* model could not be parsed by MiniZinc 2.0.1. It could be parsed by MiniZinc 1.6, and the problem is corrected in the development version of MiniZinc.

Table 2: Results for the 223 modified instances, aggregated per MiniZinc model, with the following columns: name of the model; number of instances; mean and standard deviation of the percentage of auxiliary variables; mean and standard deviation of the percentage of variables eliminated by `var-elim-idxs` (over *all* variables of the model); geometric mean and geometric standard deviation of the node rate ratio *including* preprocessing; as well as geometric mean and geometric standard deviation of the node rate ratio *excluding* preprocessing. The models are ordered by decreasing ratio excluding preprocessing (column 'ratio search').

name	inst.	% aux.	% elim.	ratio total	ratio search
l2p	5	93 (1)	73 (4)	1.27 (1.92)	2.30 (1.12)
amaze3	5	92 (1)	10 (1)	0.71 (1.98)	1.71 (2.83)
league	11	94 (4)	30 (13)	1.37 (1.71)	1.45 (1.64)
openshop	5	96 (1)	96 (1)	0.92 (2.15)	1.35 (1.13)
ship-schedule	15	82 (1)	37 (4)	0.52 (3.07)	1.32 (1.07)
wwtpp-real	10	75 (1)	70 (2)	0.27(14.75)	1.32 (1.39)
radiation	10	64 (1)	32 (1)	1.15 (1.15)	1.22 (1.07)
wwtpp-random	5	75 (0)	62 (0)	0.46 (8.31)	1.21 (1.29)
javarouting	5	88 (1)	82 (1)	1.16 (1.03)	1.18 (1.02)
solbat	30	98 (0)	16 (0)	0.65 (2.56)	1.17 (1.34)
amaze	6	55 (1)	36 (1)	1.12 (1.05)	1.16 (1.01)
project-planning	6	66 (0)	31 (2)	1.12 (1.04)	1.13 (1.04)
open-stacks	5	82 (0)	43 (2)	0.63 (2.69)	1.08 (1.04)
traveling-tppv	5	83 (0)	28 (0)	1.04 (1.01)	1.05 (1.02)
fjsp	3	75 (8)	21 (3)	0.92 (1.23)	1.04 (1.08)
tpp	6	88 (1)	24 (2)	0.97 (1.05)	1.03 (1.01)
smelt	4	72 (3)	9 (2)	1.03 (1.08)	1.03 (1.08)
train	9	53 (1)	23 (2)	1.03 (1.06)	1.03 (1.06)
pattern-set-mining	1	66 (0)	29 (0)	1.02 (1.00)	1.03 (1.00)
mspsp	2	72 (2)	54 (5)	1.02 (1.02)	1.02 (1.02)
on-call-rostering	4	66 (6)	24 (4)	1.01 (1.01)	1.02 (1.02)
carpet-cutting	2	57 (0)	37 (0)	1.01 (1.02)	1.02 (1.02)
jp-encoding	5	92 (0)	20 (0)	1.00 (1.01)	1.01 (1.01)
cyclic-rcpsp	10	91 (3)	75 (5)	1.00 (1.02)	1.00 (1.02)
rcpsp-max	6	97 (1)	96 (1)	0.99 (1.02)	1.00 (1.02)
rcpsp	4	96 (3)	94 (4)	0.99 (1.02)	0.99 (1.02)
elitserien	5	71 (0)	24 (0)	0.94 (1.07)	0.99 (1.02)
liner-sf-repositioning	4	85 (0)	10 (0)	0.97 (1.02)	0.99 (1.02)
rectangle-packing	5	88 (0)	49 (1)	0.97 (1.04)	0.98 (1.05)
stochastic-fjsp	2	81 (0)	5 (0)	0.94 (1.00)	0.98 (1.03)
still-life-wastage	5	87 (1)	5 (0)	0.88 (1.14)	0.97 (1.01)
amaze2	6	93 (0)	28 (10)	0.86 (1.17)	0.87 (1.17)
fillomino	2	94 (0)	6 (0)	0.40 (2.27)	0.86 (1.07)
roster	5	82 (0)	63 (0)	0.78 (1.42)	0.79 (1.41)
mario	10	92 (0)	22 (1)	0.56 (1.87)	0.78 (1.33)
Total	223	83 (13)	38 (26)	0.81 (2.54)	1.12 (1.38)
Total (> 1 s.)	207	83 (14)	38 (26)	1.00 (1.45)	1.12 (1.31)
Total (> 60 s.)	174	83 (14)	40 (26)	1.09 (1.27)	1.11 (1.27)

Table 3: Results for the 172 unmodified instances, with the columns of Table 2

	inst.	% aux.	% elim.	ratio total	ratio search
Total	172	80 (20)	0 (0)	0.82 (1.80)	1.00 (1.08)
Total (> 1 s.)	159	79 (20)	0 (0)	0.95 (1.13)	1.00 (1.01)
Total (> 60 s.)	110	76 (20)	0 (0)	1.00 (1.02)	1.00 (1.01)

x = 'g' refers to Gecode and 'v' to `var-elim-idxs`. We use node rates in order to have a meaningful measure for both the instances that are solved before time-out and those that are not. This assumes that the explored search trees are the same, which is discussed in Section 6. A ratio larger than 1 means that `var-elim-idxs` is faster than Gecode. We do not consider ratios between 0.97 and 1.03 to represent a significant change.

The percentage of auxiliary variables is generally very high, with an average of 83%, but on average only 70% of all the variables are auxiliary and appear in exactly two constraints. The percentage of eliminated variables varies a lot, from as little as 5% to 96%, effectively eliminating all auxiliary variables in the case of the *openshop* model. On average, `var-elim-idxs` extends Gecode with 2.3 new propagators, with a maximum of 9 propagators for an instance of the *cyclic-rcpsp* model.

The column *ratio search* shows that preprocessing generally either improves the node rate during search (ratio larger than 1.03) or leaves it almost unchanged (ratio between 0.97 and 1.03). The node rate can be more than doubled: see the *l2p* model. For the four models at the bottom of the table, however, the performance is worse after preprocessing. On average, the node rate during search is 1.12 times higher. The geometric standard deviation is generally low, i.e., close to 1.0, for instances of the same MiniZinc model, except when some of the instances are solved very fast, partly due to measurement errors.

The column *ratio total* shows that, when also counting the time for preprocessing, the results are still promising. On average, the node rate is 0.81 times lower using preprocessing. This number is strongly affected by instances that are solved very fast. If we take into account only the 207 instances that originally take more than one second to solve, then the node rate of `var-elim-idxs` is on average identical to the one of Gecode. If we take into account only the 174 instances that originally take more than one minute to solve, then the node rate of `var-elim-idxs` is on average 1.09 times higher.

Interestingly, Table 2 also shows that there is no strong correlation between the number of eliminated variables and the node rate ratio. For instance, nearly all auxiliary variables of the *rcpsp* model are eliminated but the node rate ratio is close to 1, despite the fact that the number of propagator calls is divided by two. This probably indicates that the generated indexical-based propagator suffers from some inefficiencies.

The median preprocessing time for the 223 modified instances is 4.4 seconds, roughly equally divided between the time spent by our code in Scala and the time spent by the `g++` compiler. The minimum time is 2.9 seconds, of which a

closer analysis shows that more than 2.5 seconds are actually spent in the set-up, such as loading classes or parsing header files, independently of the size of the instance or the number of identified patterns. It is important to note that neither the g++ compiler nor our indexical compiler were developed for such a use-case, as compilation is usually performed offline. The median preprocessing time for the 172 instances unmodified by preprocessing is 0.9 seconds. The minimum time is 0.7 seconds, again mostly spent in loading classes. The largest preprocessing time observed is 30 seconds, in the case of a very large *nmseq* instance. Table 3 reports aggregated results for the 172 unmodified instances: the cost of uselessly running the preprocessing is largely unnoticeable for unmodified instances that take more than 60 seconds to be solved.

6 Discussion

In the light of the experimental results, this section discusses more thoroughly the merits and limitations of our approach.

6.1 Related Work

Dealing with the introduction of auxiliary variables is an important challenge for developers of both solvers and modelling languages.

Variable Views. The initial purpose of variable views [3, 16] and domain views [19] is to reduce the number of implemented propagators in a solver, but they can also be used to eliminate some auxiliary variables. A view allows one to post a constraint on an argument that is a function of a variable instead of a variable. If a constraint is of the form $x = f(y)$, where x and y are variables, then one can introduce a view $v = f(y)$ and replace the variable x by the view v. Compared with our approach, views have the benefits that they are not limited to variables appearing in two constraints and that they do not require generating new propagators, hence that they can eliminate variables that appear, for example, in global constraints. Views are however in general limited to *unary* functions, except in [3]. More importantly, to the best of our knowledge, no solver automatically transforms a flattened constraint into a view.

Flattening and Modelling Techniques. Common subexpression elimination [13, 14] reduces the number of auxiliary variables by merging into a single variable all variables that represent the same expression. This also has the effect of increasing the amount of filtering. Hence, as explained in Section 3.1, we do not eliminate such variables.

Half-reification [5] is a modelling technique that replaces constraints of the form $B \equiv \phi$ by $B \implies \phi$, where B is a Boolean variable and ϕ a Boolean expression. Although this does not reduce the number of variables, it can reduce solving time by having simpler constraints. However, there are no half-reified

constraint predicates in FlatZinc. Our approach enables some optimisation in the spirit of half-reification, as shown in the example of Section 4.

Model globalisation [10] aims at replacing a collection of constraints at the MiniZinc level by an equivalent global constraint. Such a replacement usually reduces the number of auxiliary variables and increases the amount of filtering, provided the global constraint is not decomposed during flattening. Globalisation may improve solving time much more than our approach but it is an offline and interactive process, hence orthogonal to our online and automated approach.

Propagator Generation. Our approach uses our indexical compiler to generate propagators. The generation of stateless propagators [8] is an alternative that can yield much faster propagators. It is however limited by the size of the domains, as the constraint is essentially represented extensionally. It is meant to be used offline, as are other approaches to propagator generation, such as [4].

6.2 Properties and Extensions

Unlike most of the approaches in Section 6.1, our approach is entirely online and automated. We review here some of its properties and discuss possible extensions.

Search Tree Shape. Given a search strategy and enough time, the search trees explored by Gecode on the original model and by `var-elim-idxs` on the modified model are the same if all the following conditions are respected:

- The search strategy does not depend on the propagation order.
- The search strategy does not need to branch on the auxiliary variables.
- The generated propagators do the same filtering as the replaced propagators.

Except in the case of the *roster* model, where the search strategy is incompletely specified, the two first conditions are respected for all the instances we used in Section 5. The third condition is more difficult to check, but seems generally respected: out of the 84 modified instances that did not time out, only 7 instances had a different and always larger search tree, namely one *fjsp* instance, two *league* instances, one *roster* instance, and three *still-life-wastage* instances.

Domains. Our approach assumes that the domains of the eliminated variables are non-constraining because the shared variable X is existentially quantified without specifying a domain. This is why we restricted ourselves to variables introduced by the MiniZinc flattener, annotated with `var_is_introduced`, as the domains proved to be non-constraining for those variables. However, auxiliary variables may also be introduced manually to simplify a model. A sound way to extend our approach to such variables while retaining correctness is to verify that the domain is non-constraining before considering a variable x for replacement, by only considering how the propagators of the two constraints in which x appears reduce the domain of x given the domains of their other variables. This would also let us apply our approach to other modelling languages that do not have the `var_is_introduced` annotation, such as, e.g., XCSP [15].

Instances and Problems. We made a deliberate choice to work at the FlatZinc, rather than MiniZinc, level for two reasons. First, it is much simpler to work with a flat format than with a rich modelling language. Second, it might not be clear before or during flattening what the frequent patterns are. This choice led us to work with individual instances. However, instances from the same MiniZinc model share the same frequent patterns. Hence, when several instances of the same MiniZinc model must be solved successively, most of the results of the preprocessing of the first instance can actually be reused to reduce the preprocessing time of the following ones. In particular, when the preprocessing does not modify the FlatZinc model, detecting this on small instances saves the potentially high cost of unnecessarily parsing large instances.

Improved Propagator Generation. As seen in Table 2, `var-elim-idxs` does not remove all the auxiliary variables. Partly, this is not a limitation of our approach but of its implementation. Increasing the reach of our approach amounts to improving the generation of the propagators in order to handle efficiently more constraints, including n-ary ones. This can be done by improving our indexical compiler [11] or by using other techniques such as [8] or [3], but such improvements are orthogonal to this paper. Our experiments show that our approach is already practical as it is.

Increased Filtering. When identifying patterns, if more than one shared variable is identified, then it is possible to generate propagators achieving more filtering. Our approach can be extended to multiple shared variables. However, for it to be worthwhile, it is necessary to ensure that the propagator generation takes advantage of multiple occurrences of a variable other than X in the checker. This is currently not the case but an interesting line of future work.

7 Conclusion

We presented a new approach to eliminate many of the auxiliary variables introduced into a flattened constraint-based model. Our approach adds a preprocessing step that modifies the flattened model and extends the solver with propagators generated on the fly for new constraint predicates. This is made possible through the generation of indexical-based propagators from logical formulas. Experiments with our prototype implementation show that our approach makes a solver about 9% faster on average, and sometimes more than 2 times faster, for instances that take more than one minute to solve. This indicates that our preprocessing should be activated for instances that are difficult to solve, which are the ones for which it is important to decrease solving time.

Acknowledgements. This work is supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council (VR). We thank the anonymous reviewers for their constructive and insightful comments.

References

1. Becket, R.: Specification of FlatZinc. http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf
2. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present, and future. Constraints 12(1), 21–62 (March 2007), the catalogue is at http://sofdem.github.io/gccat
3. Correia, M., Barahona, P.: View-based propagation of decomposable constraints. Constraints 18(4), 579–608 (2013)
4. Dao, T.B.H., Lallouet, A., Legtchenko, A., Martin, L.: Indexical-based solver learning. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 541–555. Springer (2002)
5. Feydy, T., Somogyi, Z., Stuckey, P.: Half-reification and flattening. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 286–301. Springer (2011)
6. Frisch, A.M., Grum, M., Jefferson, C., Martinez Hernandez, B., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: IJCAI 2007. pp. 80–87. Morgan Kaufmann (2007)
7. Gecode Team: Gecode: A generic constraint development environment (2006), http://www.gecode.org
8. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. Artificial Intelligence 211(0), 1–33 (2014)
9. Hassani Bijarbooneh, F.: Constraint Programming for Wireless Sensor Networks. Ph.D. thesis, Department of Information Technology, Uppsala University, Sweden (2015), http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-241378
10. Leo, K., Mears, C., Tack, G., Garcia de la Banda, M.: Globalizing constraint models. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 432–447. Springer (2013)
11. Monette, J.N., Flener, P., Pearson, J.: Towards solver-independent propagators. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 544–560. Springer (2012)
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007), the MiniZinc toolchain is available at http://www.minizinc.org
13. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 590–605. Springer (2014)
14. Rendl, A., Miguel, I., Gent, I.P., Jefferson, C.: Automatically enhancing constraint model instances during tailoring. In: Bulitko, V., Beck, J.C. (eds.) SARA 2009. AAAI Press (2009)
15. Roussel, O., Lecoutre, C.: XML representation of constraint networks: Format XCSP 2.1. CoRR abs/0902.2362 (2009), http://arxiv.org/abs/0902.2362
16. Schulte, C., Tack, G.: View-based propagator derivation. Constraints 18(1), 75–107 (2013)
17. Scott, J.D.: Rapid prototyping of a structured domain through indexical compilation. In: Schaus, P., Monette, J.N. (eds.) Domain Specific Languages in Combinatorial Optimisation (CoSpeL workshop at CP 2013) (2013), http://cp2013.a4cp.org/workshops/cospel
18. Sidebottom, G., Havens, W.S.: Nicolog: A simple yet powerful cc(FD) language. Journal of Automated Reasoning 17, 371–403 (1996)

19. Van Hentenryck, P., Michel, L.: Domain views for constraint programming. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 705–720. Springer (2014)
20. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). Tech. Rep. CS-93-02, Brown University, Providence, USA (January 1993), revised version in *Journal of Logic Programming* 37(1–3):293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.