Towards Semi-Automatic Learning-based Model Transformation

Kiana Zeighami[1],
Kevin Leo[1], Guido Tack[12], and Maria Garcia de la Banda[12]

[1] Faculty of IT, Monash University, Australia
[2] Data61/CSIRO, Australia
{kiana.zeighami, kevin.leo, guido.tack, maria.garciadelabanda}@monash.edu

Abstract. Recently, [16] showed that the nogoods inferred by learning solvers can be used to improve a problem model, by detecting constraints that can be strengthened and new redundant constraints. However, the detection process was manual and required in-depth knowledge of both the learning solver and the model transformations performed by the compiler. In this paper we provide the first steps towards a (largely) automatic detection process. In particular, we discuss how nogoods can be automatically simplified, connected back to the constraints in the model, and grouped into more general "patterns" for which common facts might be found. These patterns are easier to understand and provide stronger evidence of the importance of particular constraints. We also show how nogoods generated by different search strategies and problem instances can increase our confidence in the usefulness of these patterns. Finally, we identify significant challenges and avenues for future research.

1 Introduction

Lazy Clause Generation (LCG) [11, 3] is a powerful solving technique that combines the strengths of Constraint Programming and SAT solving. It works by instrumenting finite domain propagation to record the reasons for each propagation step, thus creating an implication graph like the ones built by SAT solvers [9]. This graph is used to derive nogoods (i.e., reasons for failure) that are recorded as clausal propagators and propagated efficiently using SAT technology. The combination of constraint propagation and clause learning can dramatically reduce search and greatly improve performance (e.g., [14, 15]).

Shishmarev et al.[16] have shown that the nogoods inferred by a learning solver when executing a model may be used to design model transformations that improve its execution. In hindsight, this should have been expected: a nogood is introduced when the propagation achieved by the constraints in the model is not strong enough to avoid a failure during search. By identifying the nogoods that caused the biggest search reduction and connecting them back to the constraints in the model, Shishmarev et al. were able to improve propagation for two models. This was achieved by modifying the model constraint that generated the nogood, and by adding a new redundant constraint (suggested by the nogood).

Inferring *useful* redundant constraints for a given model is extremely difficult. Thus, using nogoods to achieve this and to strengthen the constraints already

in the model is an exciting new approach with significant potential. However, Shishmarev et al. used a manual method to infer model improvements based on the inferred nogoods. Further, the inference required in-depth knowledge of the learning solver and of the transformation performed (in this case by the MiniZinc [10] compiler) to the user's model to become the input to the target solver. In this paper we perform the first steps towards a semi-automatic process. In particular, we show how the nogoods can be automatically renamed, simplified, connected back to the constraints in the model, and grouped into more general patterns for which common facts might be found. These patterns are easier to understand by (expert) modellers than the raw nogoods, and their associated facts can help these modellers find useful modifications to the model. Additionally, we show how the generation of nogoods using different search strategies and instances of a problem can be used to increase our confidence in the usefulness of these patterns. We show the potential of the approach with two case studies, and finish by identifying some of the significant challenges with which we are still faced, and some avenues for future research that may help resolve them.

2 Background

Constraint Programming: A *constraint problem* P is a tuple (C, D, f), where C is a set of constraints, D a *domain* which maps each variable x appearing in C to a set of values $D(x)$, and f an (optional) objective function. Set C is logically interpreted as the conjunction of its elements, and D as the conjunction of unary constraints $x \in D(x)$, for each variable x appearing in C. A *literal* of P is a unary constraint whose variable appears in C. A constraint solver starts from a problem $P \equiv (C, D, f)$ and applies propagation to reduce domain D to D' as a fixpoint of all propagators for C. If D' is equivalent to *false* ($D'(x)$ is empty for some variable x), we say P is failed. If D' is not equivalent to *false* and fixes all variables, we have found a solution to P. Otherwise, the solver splits P into n subproblems $P_i \equiv (C \wedge c_i, D', f), 1 \leq i \leq n$ where $C \wedge D' \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n)$ and where c_i are literals (the *decisions*), and iteratively searches these.

The search proceeds making decisions until either (1) a solution is found, (2) a failure is detected, or (3) a restart event occurs. In case (1) the search either terminates if the model has no objective function f, or computes the value of f, sets a bound for the next value of f to be better (greater or smaller, depending on f) and continues the search for this better value. In case (2), the search usually backtracks to a previous point where a different decision can be made. In case (3) the search restarts, possibly incorporating new constraints previously learnt.

Lazy Clause Generation: LCG solvers [11, 3] extend CP solvers by instrumenting their propagators to explain domain changes in terms of *equality* ($x = d$ for $d \in D(x)$), *disequality* ($x \neq d$) or *inequality* ($x \geq d$ or $x \leq d$) literals. An *explanation* for literal ℓ is $S \rightarrow \ell$, where S is a set of literals (interpreted as a conjunction). For example, the explanation for the propagator of constraint $x \neq y$ inferring literal $y \neq 5$, given literal $x = 5$, is $\{x = 5\} \rightarrow y \neq 5$. Each new literal inferred by a propagator is recorded together with its explanation, forming an *implication graph*. Whenever the search reaches a failure, LCG solvers use the implication graph to compute a *nogood* N, that is, a set of literals (interpreted as

a conjunction) that represents the reason for the failure and cannot be extended to a solution. Then, they add its negation ($\neg N$, a set of literals interpreted as their disjunction) as a clausal propagator and backtrack, resuming the search. These learnt clauses ensure the search cannot fail again for the same reasons.

Modelling: We distinguish between a constraint problem *model*, where the input data is described in terms of parameters (i.e., variables that will be fixed before the search starts), and a particular model *instance*, where the values of the parameters are added to the model. Constraint models are usually defined in a high level language, such as Essence [4] or MiniZinc [10], and their instances are compiled into a *flattened* format, where loops are unrolled into the appropriate set of constraints, and global constraints are potentially decomposed into a representation suitable for the selected solver. Note that the compiler may introduce new variables and constraints during this flattening process, and the solver can also introduce further variables and constraints during its execution of the instance. The nogoods of an LCG solver will consist of literals that refer to this fully flattened and decomposed instance of the problem. This makes their analysis in terms of the high-level model particularly challenging.

Paths: We use the concepts of *variable paths* [6] and *constraint paths* [7], which assign a unique identifier to each variable and constraint appearing in a flattened instance. They allow us to connect the flattened variables and constraints to the model's source code. Each identifier describes the path the compiler took when compiling a MiniZinc instance (that is, a MiniZinc model and its input data) to FlatZinc, from the actual model to the point where a new variable or constraint is introduced. For example, the following is a (simplified) constraint path:

`14:12-15:61 forall:p1=1,p2=6` `14:36-15:60 ->` `14:37-14:74 /\` `14:37-14:74 clause`

Each of its components has two parts: four numbers denoting the span of text in the MiniZinc model that the expression came from (with format from line:column-to line:column); and a textual description of what the expression represents. The above path represents a `clause` that was inserted into the final FlatZinc, as a result of encoding a negated (thus the `clause` part) conjunction `/\` appearing in the left hand side of the implication (`->`) that appears in the `forall` loop from lines $14 - 15$, with index variables `p1=1` and `p2=6`.

3 Towards Automation: the `freepizza` case

Shishmarev et al. [16] demonstrated how the clauses learnt by an LCG solver can be used to improve a model. To achieve this, they executed the model using the LCG solver Chuffed [2], replayed its search decisions (in the same order) using the non-LCG solver Gecode [13], merged the two resulting search trees, counted the number of nodes explored by Gecode that were not explored by Chuffed, and assigned them to the learnt clause(s) that helped Chuffed fail before Gecode. This formed a ranking of all clauses based on the total number of search nodes avoided by each clause, with the top-ranked clauses being the most effective for reducing search. Finally, they manually inspected the 10 most effective clauses to learn information that could help improve the model.

This section introduces several techniques for automating part of the process described in [16] using MiniZinc (the techniques can, however, be applied in other

modelling languages). We use one of their case studies (free pizza) to illustrate the manual process on a hard problem (Chuffed only solved 1 of 5 instances in the 2015 MiniZinc Challenge [17]), and the issues faced when automating it. In this problem customers get pizzas either by paying for them or by using vouchers. Each voucher (a, b) allows customers to get b number of pizzas for free if they pay for a number of pizzas, and none of the b pizzas are more expensive than the a ones. A customer who has m vouchers and wants n pizzas aims to minimise the amount paid for the n pizzas. Their MiniZinc model (called `freepizza`) is:

```
1  int: n;    set of int: PIZZA = 1..n;     % number of pizzas wanted
2  array[PIZZA]   of int: price;            % price of each pizza
3  int: m;    set of int: VOUCH = 1..m;   % number of vouchers
4  array[VOUCH] of int: buy;               % buy this many to use voucher
5  array[VOUCH] of int: free;              % get this many free
6
7  set of int: ASSIGN = -m .. m; % i -i 0 (free/paid with voucher i or not)
8  array[PIZZA] of var ASSIGN: how;
9  array[VOUCH] of var bool: used;
10
11 constraint forall(v in VOUCH)(used[v]<->sum(p in PIZZA)(how[p]=-v)>=buy[v]);
12 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
13 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
14 constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
15                                   -> price[p2] <= price[p1]);
16 int: total = sum(price);
17 var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);
```

Lines 1–5 introduce the parameters: numbers n and m, an array for the pizzas' prices, and two arrays for vouchers s.t. `(buy[i],free[i])` represents the ith voucher (a, b). The next three lines define two arrays of decision variables: `used[v]`, which is true iff voucher `v` was used; and `how[p]`, which is `v` if pizza `p` was free thanks to voucher `v`, is `0` if `p` was paid for and not used in any voucher, and is `-v` if `p` was paid for and used to get free pizzas with voucher `v`.

Constraints start in line 11, which states that if voucher `v` was used, then the total number of pizzas bought and assigned to `v` must be greater than or equal to the number of pizzas required by it. Line 12 states similar information but in the opposite direction: the total number of pizzas bought and assigned to voucher `v` must be less than or equal to `used[v]*buy[v]`. Together they constrain the total number of pizzas bought for `v` to be equal to `buy[v]`, if used. The constraint in line 13 states that the total number of free pizzas obtained thanks to voucher `v` must be smaller than or equal to the number of free pizzas allowed by `v` if used (`used[v]*free[v]`). The last constraint states that if there are two pizzas `p1` and `p2` assigned to the same voucher with `p2` being free and `p1` being paid for (given `how[p1] < how[p2]` and `how[p1] = -how[p2]`), then the price of `p2` must be lower than or equal to that of `p1`. Finally, the objective function is defined as the sum of the prices of the pizzas that are bought.

Table 1 shows the top 10 clauses found in [16] for `freepizza` with data:

```
n = 10; m = 4; price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
buy = [1, 2, 3, 3]; free = [1, 1, 2, 1];
```

Each clause is interpreted as the disjunction of its literals. For example, the clause ranked 4th, {`how[5]`$\neq$`-3`,`how[3]`$\neq$`3`}, states that pizza 3 cannot be obtained for free using voucher 3 by paying for pizza 5 with that voucher (this might be easier to see in its equivalent form $\neg$(`how[5]=-3` $\wedge$ `how[3]=3`)).

Table 1. Taken from [16]: Most effective learnt clauses in `freepizza`

Rank	Reduction	Clause
1	3425	how[1]=-1 how[2]=-1 how[3]=-1 how[4]=-1 how[5]=-1 how[1]=-2 how[2]=-2 how[3]=-2 how[4]=-2 how[5]=-2 how[6]≤0 how[6]≥3
2	2068	how[7]≤2 how[7]≥4 how[1]≠-3 how[1]≥-2
3	1712	how[4]≠3 how[1]=-3 how[2]=-3 how[3]=-3 how[4]=-3
4	1636	how[5]≠-3 how[3]≠3
5	1636	how[8]≠-3 how[3]≠3
6	1636	how[9]≠-3 how[3]≠3
7	1636	how[10]≠-3 how[3]≠3
8	1489	how[6]≤2 how[6]≥4 how[1]≠-3 how[1]≥-2
9	1404	how[5]≠-3 how[4]≠3 how[4]≤2
10	1403	how[10]≠-3 how[4]≠3

3.1 Renaming literals

The clauses in Table 1 are already the result of significant manual interpretation and analysis. For example, the 4th clause came from renaming (and simplifying) solver-level clause {`X_INTRODUCED_4_`≠-3, `X_INTRODUCED_2_`≠3, `X_INTRODUCED_2_`≤2}, where the names `X_INTRODUCED_4_` and `X_INTRODUCED_3_` were introduced by the MiniZinc compiler for the model variables in array positions `how[5]` and `how[3]`, respectively. Thus, our first step towards automation is to transform clauses to refer to model-level variables and expressions. For simple renamings as in the example above, we could instrument the MiniZinc compiler to keep a map from solver-level to model-level names. New variables may, however, also be introduced when flattening expressions. For example, the compiler may introduce an auxiliary variable for the result of an addition, or the result of flattening a `let` expression. Connecting such variables back to model-level names is more complex. We propose to do this by using variable paths [6]. Consider, for example, the literal `X_INTRODUCED_244_=true`, which appears in one of the clauses generated by Chuffed for `freepizza`. As variable `X_INTRODUCED_244_` does not correspond directly to any model-level variable, the compiler produces the following path:

14:12-15:61 `forall:p1=1,p2=6` | *14:36-15:60* `->` | *14:37-14:74* `/\` | *14:37-15:74* `clause` | *14:58-14:74* `=` | *14:58-14:74* `int_lin_eq`

The final entry in the path shows the location in the model of the expression that corresponds to `X_INTRODUCED_244_`: line 14, columns 58–74, which has expression `"how[p1]= -how[p2]"`. The path also shows that the expression is located within the `forall` that spans lines 14:12-15:61, within the implication (`->`) that spans lines 14:36-15:60, within the conjunction (`/\`) that spans line 14:37-14:75, within the `clause` that the negated conjunction was encoded as, and within the = that spans line 14:58-14:74. Since the path ends with an `int_lin_eq` constraint (integer linear =), we can deduce that the introduced variable is the Boolean control variable for the reified version of expression `"how[p1]= -how[p2]"` in the model. Since the path also records the values of loop variables `p1` and `p2`, these can be automatically substituted, yielding `"how[1]= -how[6]"=true`.

3.2 Simplifying literals and clauses

The readability of any clause can be further improved by simplifying its literals based on their semantics. We do this in two different ways. First, we simplify literals whose variables were introduced by the compiler, and which correspond to Boolean expressions in the model, as follows:

> **positive:** if the right-hand side of a literal is true (e.g. `=true, =1, >=1`), and the left-hand side is a Boolean expression of the form $e_1 \, op \, e_2$, where *op* is a binary operator, then we can simply remove the right hand side.
> **negative:** if the right-hand side of a literal is false (e.g. `=false, =0, <=0`), and the left-hand side is a Boolean of the form `x` *op* `v`, where `x` is a variable, *op* is a binary operator and `v` is an integer value, then we can negate the left hand side and remove the right hand side.

For example, literal `"how[1] < how[6]"=true` becomes `how[1] < how[6]`, and literal `"how[1]=-1"<=0` (which comes from the sum of reified equality constraints in line 12 of the model) becomes `how[1]`$\neq$`-1`. Simplifying literals with more complex left-hand sides (i.e., arbitrary MiniZinc expressions), requires a deeper integration with the MiniZinc compiler and is left for future work.

Second, we eliminate from each clause any literal that entails (i.e., implies) other literals in the clause, since $A \vee B \vee C$ is equivalent to $A \vee B$, if C entails B. This makes the clause easier to understand and, as shown in Section 3.4, makes it easier to automatically detect clause patterns. We automatically simplify a clause by applying the following rules to literals that operate on the same variable x:

- $\neq$ a literal of the form $x \neq v$ is entailed by any other literal $x = v'$ such that $v \neq v'$ and any literal $x \leq v'$ ($x \geq v'$) such that $v' \leq v$ ($v' \geq v$). Thus, those other literals are eliminated. For example, clause $\{x = 1, x \leq 1, x \geq 4, x \neq 2, \dots\}$ becomes $\{x \neq 2, \dots\}$.
- $\geq$ ($\leq$) a literal of the form $x \geq v$ ($x \leq v$) is entailed by any other literal $x \geq v'$ ($x \leq v'$) s.t. $v < v'$ ($v > v'$). Thus, we only keep the literal with the lowest (highest) bound. For example, clause $\{x \leq 1, x \leq 2, x \geq 3, x \geq 4, \dots\}$ becomes $\{x \leq 2, x \geq 3, \dots\}$.
- $\leq\geq$ if a clause contains two literals $x \leq v_1$ and $x \geq v_2$, s.t. $v_2 - v_1 = 2$, then these two literals can be replaced by a single literal $x \neq v_2 - 1$. For example, clause $\{x \leq 1, x \geq 3, \dots\}$ becomes $\{x \neq 2, \dots\}$.

Applying these rules to our clause {`how[5]`$\neq$`-3, how[3]`$\neq$`3, how[3]`$\leq$`2`} yields the one ranked 4th in Table 1: {`how[5]`$\neq$`-3`, `how[3]`$\neq$`3`}, since `how[3]`$\leq$`2` entails `how[3]`$\neq$`3`, and is thus eliminated by the $\neq$ rule.

Note that all simplifications are done after renaming the introduced variables. This is useful, as literals that reference different introduced variables may later become a single literal. This happens often in `freepizza` as, for example, the `sum` function expects integer variables, but in the model it is given Booleans. Each Boolean variable is coerced by the compiler to be integer by introducing a new integer variable and posting a `bool2int` predicate equating the Boolean to the integer one. For expression `sum(p in PIZZA)(how[p]=-v)`, both the original Boolean variables and the introduced integer variables refer to

the expression `"how[p]=-v"` in the model. However, before renaming, the literals may appear different (e.g., `X_INTRODUCED_45_=false` and `X_INTRODUCED_53_<=0`) even though they mean the same thing (`how[p]`$\neq$`-v`). By performing the simplification after the renaming, both variables are known to be (or come from) a Boolean expression, and are thus simplified to `how[p]`$\neq$`-v`. Interestingly, if these simplifications had been applied to the clauses in Table 1, Shishmarev et al. would have realised that clauses 2, 8 and 9, can be further simplified to {`how[7]`$\neq$`3`, `how[1]`$\neq$`-3`}, {`how[6]`$\neq$`3`, `how[1]`$\neq$`-3`}, and {`how[5]`$\neq$`-3`, `how[4]`$\neq$`3`}.

3.3 Connecting clauses to the constraints in the model

One of the main achievements of Shishmarev et al. was to use the learnt clauses to identify the need to strengthen the constraint in line 14 of the model. This need was discovered by realising that some of the top clauses were direct consequences of a single constraint; the one in line 14. To do this, Shishmarev et al. manually linked the learnt clauses (or more accurately, their literals) to the model constraints they were derived from.

To automate this step, we instrumented Chuffed to record the solver-level constraint directly responsible for adding any literal to the clause database. That is, for each explanation $S \rightarrow \ell$ added by a constraint with identifier id_c, we record that ℓ was directly generated from id_c. Then, when a clause Cl is generated, we obtain the constraint identifier of each literal ℓ in Cl, ask that constraint to provide us with the explanation S that was used to generate ℓ, and recursively apply the same method for all literals in S. This allows us to (lazily) trace back all constraints involved in generating the literals of Cl.

For clause {`how[5]`$\neq$`-3`, `how[3]`$\neq$`3`}, our method identifies a single solver-level constraint as responsible for all literals. Using constraint paths, we can trace it back to the expression `how[p1]=-how[p2]` on line 14 of `freepizza`, with loop variables `p1=5` and `p2=3`. Thus, our automatic method successfully identifies the line in which the constraint responsible for the clause appears. While connecting the clauses to the model constraints via a textual path is already very useful, it would be easier for users to get a visual connection. To achieve this, we have modified the MiniZinc IDE to visually highlight the parts of the user's model responsible for a given clause (see Figure 1).

3.4 Finding patterns among clauses

At this point, our automatic method can make a clause clearer, simpler and visually connect it to the constraints it came from. However, a single clause may not be worth exploring, even if it led to a considerable search reduction. The modeller may need stronger evidence to start what can be a lengthy exploration (as clauses are in practice quite complex, even after renaming and simplification). The evidence can be much stronger if several clauses with a similar *pattern* can be found to have significantly reduced the search. For example, Shishmarev et al. [16] focused on clause 4 ({`how[5]`$\neq$`-3`,`how[3]`$\neq$`3`}) not only because it was one of the top clauses and it was short (and thus easier to understand), but importantly, because it was part of several *similar clauses* in the top 10 (which they identified as clauses 5, 6, 7, and 10).

```
int: n;    set of int: PIZZA = 1..n;     % number of pizzas
array[PIZZA]   of int: price;            % price of each pizza
int: m;    set of int: VOUCHER = 1..m;   % number of vouchers
array[VOUCHER] of int: buy;              % buy this many to use voucher
array[VOUCHER] of int: free;             % get this many free

set of int: ASSIGN = -m .. m; % i -i 0 (pizza is free/paid with voucher i or not)
array[PIZZA] of var ASSIGN: how;
array[VOUCHER] of var bool: used;

constraint forall(v in VOUCHER)(used[v]<->sum(p in PIZZA)(how[p] = -v) >= buy[v]);
constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
                                   -> price[p2] <= price[p1]);
int: total = sum(price);
var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);
```

Fig. 1. Constraint responsible for learnt clause {how[5]≠-3, how[3]≠3}

It is easy to show that these five clauses share the same *pattern*: {how[*A*]≠-*B*, how[*C*]≠*B*}, where *A* and *C* are pizzas, and *B* is a voucher. For example, clause 4 can be obtained by mapping *A*/5, *C*/3, and *B*/3. In fact, with the simplifications provided above, all clauses in Table 1 except 1 and 3 can be shown to share this pattern *and* to come from the same constraint. Grouping clauses with the same pattern can help modellers in two ways: (a) while individual clauses may not seem to contribute much to the overall search reduction, a group of clauses with the same pattern may do so; and (b) a pattern may identify a general constraint, i.e., something that is true for a whole range of parameters, and may therefore suggest a redundant constraint that can be added to the model.

We define a *pattern* to be the most specific (or least) generalisation [12] (MSG) of a set of clauses. To automatically obtain patterns, our method first renames and simplifies each clause. Then, it sorts the literals in each clause by variable name, operator and constant value. Finally, for all clauses that have the same sequence of literal operators, and the same sequence of variable types, it computes the MSG for all its subsets. For example, the following two (renamed, simplified and sorted) clauses for our `freepizza` instance:
{how[1]=-1,how[2]=-1,how[3]=-1,how[4]=-1,how[6]≠1} and
{how[1]=-2,how[2]=-2,how[5]=-2,how[7]=-2,how[6]≠2}, have the same sequence of literal operators (=,=,=,=,≠) and the same sequence of variable types (how[_],how[_],how[_],how[_],how[_]). Our method computes the pattern {how[1]=-A,how[2]=-A,how[B]=-A,how[C]=-A,how[6]≠A}, which can be mapped back to the clauses by applying the name/constant maps {A/1,B/3,C/4} and {A/2,B/5,C/7}, respectively, where {B,C} are known to be pizzas and {A} a voucher. However, neither of the two clauses would form a pattern with {how[1]=-2,how[2]=-2,how[6]≠2}, as they have a different number of literals, or with {how[1]=-1,how[2]=-1,how[3]=-1,used[1]=true,how[6]≠1}, as they have different variable types.

Formally, each clause *Cl* is stored as a tuple (*Id*, *SeqLits*, *Red*, *Cons*) containing the clause identifier *Id*, its renamed, simplified and sorted sequence of literals *SeqLits*, its associated search reduction *Red*, and the constraints *Cons* that generated it. A pattern *Pt* is stored as a tuple (*PId*, *PSeqLits*, *Maps*, *PRed*, *PCons*) containing the pattern identifier *PId*, its sequence of literals *PSeqLits*, a set *Maps* of tuples of the form (*Map*, *Id*), where applying *Map* to *PSeqLits* yields the sequence of literals in the clause identified by *Id*, the total search reduction

PRed achieved by its clauses (the sum of the *Red* of each clause identified by *Maps*), and the set of constraints *Cons* that generated any of its clauses (the union of the *Cons* of each of the clauses identified by *Maps*).

Note that a clause may appear in many different patterns. In fact, this will often be the case, as our algorithm can be seen as computing a pattern for each subset of the set of (renamed, simplified and sorted) clauses that have the same sequence of (a) literal operators and (b) variables types. The algorithm starts with the set of renamed, simplified and sorted *Clauses* computed for a given instance as described in previous sections, and an empty set of *Patterns*. Then, for every clause $Cl = (Id, SeqLits, Red, Cons)$ in *Clauses*, it does the following:

1. Transform Cl into pattern $Pt = (PId, SeqLits, \{([], Id)\}, Red, Cons)$, where PId is a new identifier.
2. Set *NewPatterns* to $\emptyset$
3. For each pattern $Pt' = (PId', PSeqLits, Maps, PReds, PCons)$ in *Patterns* with the same sequence of literal operators and variable types as *SeqLits*:
   (a) Find a most specific generalisation $PSeqLits'$ for *SeqLit* and *PSeqLits*, with associated mapping *Map*.
   (b) Add $(PId', PSeqLits', Maps \cup \{(Map, Id)\}, PRed + Red, PCons \cup Cons)$ to *NewPatterns*.
4. Merge *Patterns* and *NewPatterns*.
5. Add Pt to *Patterns*.

Note that while the pattern Pt added in step 5 above is known to be different from all others in *Patterns* (since Pt is essentially a clause and no two clauses are identical), it is possible to have a pattern P_1 in *NewPatterns* with a sequence of literals identical (up to variable renaming *Ren*) to pattern P_2 in *Patterns*. We can detect this when merging *Patterns* and *NewPatterns* in step 4, and simply merge P_1 and P_2 (by first applying *Ren* to them, and then computing the unions of their mapping and constraint sets, and summing up their reductions).

Once all patterns are computed, our method ranks the results by the amount of reduction associated with each pattern and presents it to the modeller. This allows modellers to notice clauses that may not result in significant search reductions when considered individually, but do when considered together.

3.5 Inferring facts for clause patterns

While patterns are a clearer way to condense the information provided by several clauses, the insights obtained by Shishmarev et al. included information that was not present in the clauses. For example, for clause {how[5]≠-3,how[3]≠3} they considered the fact that pizza 3 is more expensive than pizza 5, and for {how[1]=-1,how[2]=-1,how[3]=-1,how[4]=-1,how[5]=-1,how[1]=-2, how[2]=-2,how[3]=-2,how[4]=-2,how[5]=-2,how[6]≤0,how[6]≥3}, the fact that pizza 6 is more expensive than any other pizza in the clause.

When generalising a set of clauses to a pattern, these additional facts can serve as *conditions* under which the pattern is indeed a valid (implied) constraint. For example, let us again consider the pattern {how[*A*]≠-*B*,how[*C*]≠*B*} or, in implication form, how[*A*]=-*B* → how[*C*]≠*B*. This pattern is clearly not valid for arbitrary values of *A*, *B*, and *C*. However, for all clauses that contributed

to the pattern (including clauses 2,4-10), it is true that pizza A is cheaper than pizza C. This yields a valid constraint: if pizza A is cheaper than pizza C, then for any voucher B, if we use A with voucher B, then we cannot get pizza C for free with voucher B. We have thus inferred the following constraint, which can be added to the model (expressed as the contrapositive of the statement above to make it more similar to the original formulation):

```
constraint forall(A,C in PIZZA, B in VOUCH) ((how[A]=-B /\ how[C]=B)
                                              -> price[C]<=price[A]);
```

This constraint is logically equivalent to line 14 of the model, but it provides stronger propagation, since it explicitly states that we cannot get pizza C for free with voucher B, rather than stating this requirement indirectly. The manual analysis in [16] resulted in a different reformulation:

```
constraint forall(A,C in PIZZA) ((0 < how[C] /\ how[A]= -how[C])
                                 -> price[C] <= price[A] );
```

While they are equivalent, the automatically derived constraint is at a lower level of abstraction, as it is stated for each individual voucher `B`, rather than as a general statement on the `how` variables. An interesting direction for future research is how to enable the automatic approach to perform such generalisations.

Automating the extraction of these facts is challenging, since it requires finding relations in the model's parameters that are true for a set of clauses. We have implemented an approach that extracts *relevant* facts for a certain pattern. A fact is relevant for a pattern if it concerns objects from it, e.g., if it relates the prices of pizzas mentioned in a pattern, or the `buy` and `free` values of a voucher mentioned in a pattern. The algorithm proceeds as follows for each pattern:

1. Extract the objects in the pattern; e.g., {`how[A]`$\neq$`-B`, `how[C]`$\neq$`B`} refers to pizzas `A` and `C`, and voucher `B`.
2. Extract parameter values relating to all objects referred to by the clauses in the pattern. For the example above, infer prices for all pizzas `A` and `C` in any clause of the pattern, and `buy`/`free` information for all vouchers `B`.
3. For the extracted parameters, infer facts as simple unary relations on the objects, such as $p[X] = 0$, $p[X] < 0$ or $p[X] = \max(p)$, for each parameter p that is indexed by the type of object X. When there are two objects X and Y of the same type in a pattern, additionally infer facts as binary relations such as $p[X] = p[Y]$, $p[X] \neq p[Y]$, $p[X] \leq p[Y]$, $p[X] = -p[Y]$.

For the example with the most expensive pizza 6 above, we infer the fact that `price[A]=max(price)` if a pattern exists where pizza `A` is the most expensive one. For the example comparing two pizzas, we will infer that `price[C] > price[A]` for all clauses in the pattern {`how[`A`]`$\neq$`-`B,`how[`C`]`$\neq B$}. Since patterns may contain many clauses, the extracted facts are unlikely to always be true for all clauses. We therefore compute, for each extracted fact, the *percentage* of clauses in the pattern that it matches, and only report facts that match above a certain threshold. This is clearly a very incomplete and costly method, and it will be interesting to pursue further research into how more general facts can be extracted automatically in a reasonable amount of time.

3.6 Finding patterns across searches and across instances

Finding a pattern whose cumulative search reduction is significant for a given instance, provides the modeller with some confidence regarding the importance of the pattern and the possibility of generalising it to a model constraint. Confidence will be stronger if clauses with the same pattern appear in different instances of the same model, as this suggests the pattern and its associated information holds across different input data and is, thus, more likely to be a general property of the model. Confidence would also be stronger if the pattern significantly reduced the search across different searches, as this would indicate the associated model modification may lead to speed-ups for all those searches.

To achieve this we have implemented a simple algorithm that, for each instance, takes the union of all clauses obtained by the given set of searches, and then finds the patterns (and facts) associated to them. For each identified pattern, it then computes the percentage of instances that have produced this pattern (up to renaming). Applying this method to the results obtained with several instances and searches of the `freepizza` model, we found that the pattern `{how[A]≠B,how[C]=-B,how[D]=-B,how[E]=-B,how[F]=-B,how[G]=-B}` appears in at least two instances and most of their searches, with information `price[A]>price[C],price[A]>price[D],price[A]<price[E],price[A]<price[F]`, and `price[A]>price[G]`, having percentages between 79% and 55%. The pattern states that if pizza A is free with voucher B, then one of the other pizzas needs to be paid for with the same voucher. It is a weaker version of clause 1 in Table 1 and indicates that pizza `A` can never be obtained for free by paying the cheaper pizzas. The fact it appears so often suggests exploring a new constraint that, for each pizza p, considers the subset of pizzas cheaper (or more expensive) than p.

Note however that the absence of a clause in a particular execution does not mean the clause (and thus its associated pattern) is not valid. We therefore need to be somewhat cautious when considering the importance of the patterns obtained by the above method. Alternatively, we could modify our method to do something similar to that of [8], by testing for each pattern Pt that does not appear in some of the executions, whether at least one of its instantiations holds.

4 Preliminary Case Studies

While automating the method, we discovered new examples where the approach is applicable. This section explores two such cases.

4.1 Case Study 1: Redundant constraints

Our semi-automated approach allowed us to discover a redundant constraint that can be added to a model of the Grid Colouring problem. A grid colouring is a colouring of an $n \times m$ grid `x`, where no sub-rectangle of the grid has all of its corner cells assigned the same colour. A simplified version of the MiniZinc model for this problem is presented below. The loop on lines $4-8$ enumerates all sub-rectangles of the grid, and states that at least one pair of orthogonally adjacent corners must be assigned distinct colours.

```
1  int: n; int: m; % Width and height of grid
2  array[1..n,1..m] of var 1..min(n,m): x;       % Colour in each cell
3
4  constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)(
5                    (   x[i,k]!=x[i,l] \/ x[i,l]!=x[j,l]
6                     \/ x[j,k]!=x[j,l] \/ x[i,k]!=x[j,k]) );
7
8  solve minimize max(x); % Number of colours used
```

Looking at the clauses produced for instances of this problem, we found a frequent, high-ranking pattern with literals {x[A,B]≠x[A,C],x[A,B]≠x[D,B], x[A,C]≠E,x[D,B]≠E,x[D,C]≠E}. One interpretation of the pattern states that if corner x[A,B] is the same colour as x[A,C] and x[D,B], then one of x[A,C],x[D,B] or x[D,C] must be assigned a different colour. Upon examination, it became clear that the constraints in the model only indirectly compared diagonally adjacent corners. To address this weakness, we added the following constraints:

```
constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)
                ((x[i,l]=x[j,k] /\ x[i,l]=x[j,l]) -> (x[i,k]!=x[i,l]))
```

These constraints did not improve Chuffed's performance but did improve Gecode's significantly. Table 2 shows Gecode's solve time (in seconds) and node count for several instances of the original model and the modified one. The number of extra constraints result in Gecode spending more time at each node but the added propagation leads to faster solve times.

4.2 Case Study 2: Strengthening model constraints

Our method also helped us discover a constraint in the time-changing graph colouring problem model that could benefit from domain (rather than the commonly used bounds) consistency [1]. In this problem, the given initial colouring of a graph must be transitioned to a given final colouring. The transition requires a certain number s of steps, each performing at most k modifications to the colours while maintaining a valid graph colouring. The objective is to first minimise the number of steps required, and then the number of modifications.

The following shows an extract from the MiniZinc model we used, which appeared in the annual MiniZinc Challenge as tcgc2[3]. The model takes as input, among others, the maximum number k of transformations per step, and sets the

[3] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/tc-graph-color

		Original		Modified	
n	m	time (s)	nodes	time (s)	nodes
5	5	0.10	34,987	0.07	5,452
5	6	0.13	35,223	0.09	5,468
6	6	0.55	131,661	0.29	16,773
6	7	1.53	9,484,042	0.77	37,727
7	7	65.47	11,565,900	23.15	904,148

Table 2. Gecode's solving time for different instances of the Grid Colouring problem.

maximum number of steps `max_s` to 10. It has an array of decision variables, where `a[i,n]=j` represents the colour `j` of node `n` in step `i`, and a decision variable `s` representing the final number of steps. The constraint displayed states that in every non-final step `i`, the sum of all transformations must be less or equal to `k`.

```
int: k;                                  % maximum colour changes per step
int: max_s = 10;                         % maximum number of steps
array [STEPS,NODES] of var COLORS: a;    % Colours of nodes at each time step
var 2..max_s: s;                         % final number of steps
...
```

```
51  constraint forall(i in 1..max_s-1)
52              (i < s -> sum (n in NODES) ( a[i,n] != a[i+1,n] ) <= k);
53  ...
```

Our method identified pattern {`a[A,B]`$\neq$`a[A+1,B]`, `a[A,B]=C`, `a[A+1,B]`$\neq$`C`} as interesting, since its clauses are highly ranked across different searches of different instances and, when combined, are responsible for a large search reduction. Further, the pattern is short (and thus easy to understand) and all its clauses come from a single model constraint (which often indicates lack of expected propagation by the constraint). Upon examination, we felt the pattern stated something so simple it should have already been captured by the propagation of the model constraints. The first literal, derived from the expression highlighted on line 52, represents the result of a reified not-equals constraint. The literal is true when the variables take different values. If this is false, the remaining literals state that either the variables both take the value `C` or they take some other value (this becomes clear when presented in implication form: {`a[A,B]=a[A+1,B]` $\wedge$ `a[A,B]`$\neq$`C` $\rightarrow$ `a[A+1,B]`$\neq$`C`} and {`a[A,B]=a[A+1,B]` $\wedge$ `a[A+1,B]=C` $\rightarrow$ `a[A,B]=C`}). While this information is obvious, the fact that the associated nogoods were being reported by the solver indicated propagation was not performing as expected. We then realised that the implementation of a reified not-equals constraint in Chuffed is bounds consistent rather than domain consistent. To resolve this, we manually modified the model to add the following information:

```
48  array[1..max_s-1,NODES] of var bool: aa;
49  constraint forall (i in 1..max_s-1, n in NODES)
50              (aa[i,n] <-> forall (c in COLORS) (a[i,n]=c <-> a[i+1,n]=c));
51  constraint forall (i in 1..max_s-1)
52              (i < s -> sum (n in NODES) ( (not aa[i,n] ) ) <= k);
53  . . .
```

Table 3 shows solve times for instances of the original and modified model. Three different search strategies with a 5 minute time limit were executed. The strategies were the fixed strategy defined in the model, Chuffed's free search strategy, and, one that alternated between the two. The results show the constraint can improve Chuffed's solving performance on some (but not all) instances of the problem. For a traditional CP solver with a domain consistent reified not-equals constraint, these extra constraints will slow down propagation, and an annotation indicating the need for domain propagation would be preferred. This shows the need for the modeller's input.

	Fixed		Free		Alternate	
Instance	Original	Modified	Original	Modified	Original	Modified
k5_5	48.48	**40.54**	**83.68**	86.20	**66.65**	86.62
k9_39	∞	∞	∞	∞	**262.28**	295.16
k10_31	∞	∞	∞	∞	78.35	**63.81**
k10_34	23.66	**21.18**	**80.65**	91.60	77.57	**69.59**
k10_41	**1.67**	1.99	**17.80**	19.60	**3.15**	3.25

Table 3. Solving times for different instances of `tcgc2` using different search strategies.

5 Status and Limitations

The semi-automatic method presented in the previous sections is a significant improvement over the manual method presented in [16], making it much easier and effective to explore the impact of the clauses learnt by a learning solver, and the possible model modifications these clauses suggest. However, the current implementation still has important limitations that are worth exploring. Here we present some of these limitations and point to possible ways to address them.

Variable Paths: Our approach for reconstructing expressions based on paths is purely textual. Thus, the paths for variables added by the compilation of a function/predicate call currently appear connected to the names of the parameters or the local variables of the function/predicate. For example, a decomposition of the `alldifferent(array[int] of var int: x)` predicate (from the MiniZinc library) will contain expression `x[i]!=x[j]`. If the model contains a call to this predicate, such as `alldifferent(y)`, our purely textual approach would not be able to rename `x` to `y` in any clauses resulting from this constraint. Therefore, modellers cannot distinguish between different invocations of the same predicate.

We plan to address this limitation by using the MiniZinc compiler to hoist local variables and parameters to be expressed in terms of the top level model variables where possible, to apply simplification rules, and to deduce accurate information about the types of expressions we have extracted.

Finding patterns: Our current implementation has three main limitations. First, to compute the patterns of a given model we must manually provide the names of the decision variables that can appear in the literals, their possible values and their types. This is needed to correctly (a) parse the clauses produced either by Chuffed or by our path-based renaming process, and (b) obtain the most specific generalisation for two clauses. This manual step can be resolved by integrating our method with the MiniZinc compiler, so that the required information can be extracted directly from it. Second, our current implementation can only handle very simple expressions in the clause literals, even though the paths for introduced variables can result in complex ones.

Finally, and importantly, the current implementation of the most specific generalisation of a pattern is very limited, as it (a) only matches sequences of literals that have the same sequence of operators and variable types, and (b) only matches the ith literal in each pair of clauses. Due to (a) we might miss

a relationship between clauses that have different numbers of literals but share a *parametric* pattern. For example, clauses `{w[1]<3,w[2]<3}`, `{w[1]<3,w[2]<3,w[3]<3}` and `{w[1]<3,w[2]<3,w[3]<3,[w4]<3}`, share the pattern `{w[x]<3| x ∈ 1..n}`. Due to (b) we might miss patterns that were obscured by the sorting or renaming of literals. For example, for clauses `{w[10]=1,w[20]=2}` and `{w[40]=2,w[50]=1}` we currently only find a pattern with sequence of literals `{w[A]=B,w[C]=D}`, and maps `{A/10,B/1,C/20,D/2}` and `{A/40,B/2,C/50,D/1}`, while matching the first and second literal of each clause would yield a more specific pattern with sequence: `{w[A]=1,w[B]=2}` and maps `{A/10,C/20}` and `{A/50,C/40}`. More sophisticated algorithms (such as the anti-unification of [5]) can handle different numbers of literals or matching any literals with the same operator. However, given the computational cost, it might only be worth it for top-ranking clauses.

Finding facts: This is one of the most difficult and interesting areas for future research. As described above, our current implementation blindly collects all simple relations between the parameter values that can be associated to objects appearing in a clause. While the resulting information is a small subset of the knowledge space, it is still costly to compute. Furthermore, other knowledge may be missed (e.g., arbitrary parametric expressions not explicitly present in the original constraints), which could help modellers understand the origin of the clause and point to better ways to improve the model.

6 Conclusions

This paper presented improvements to and steps towards the automation of the methods presented in [16]. In particular, we have shown how the literals in learnt clauses can be automatically renamed, simplified and both textually and visually tied back to the parts of the model they originated from. This will make it easier for modellers to understand the clauses. We have then shown how the resulting clauses can be grouped by generic patterns, and how to derive information regarding the objects present in these patterns. Further, we have shown how these patterns can be inferred across searches and across instances. This will help modellers determine the likelihood of a pattern being generalisable to the model to either strengthen a constraint or as a new, redundant constraint. We have also shown how the information associated with the patterns will help modellers determine the particular strengthening that should be applied, or the new constraint that should be added. We have provided two new case studies that show the applicability of the approach. Finally, we have identified several significant challenges our method still faces, and potential avenues for future research. We believe that by further automating this methodology, and potentially taking advantage of other learning solvers (such as cuts in MIP solvers), we will be able to provide tools that significantly simplify the task of finding model refinements for users of the MiniZinc toolchain.

Acknowledgements. This research was partly sponsored by the Australian Research Council grant DP180100151.

References

1. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Australasian Joint Conference on Artificial Intelligence. pp. 49–58. Springer (2006)
2. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
3. Feydy, T., Stuckey, P.J.: Lazy Clause Generation Reengineered. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 352–366. Springer (2009)
4. Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
5. Kutsia, T., Levy, J., Villaret, M.: Anti-unification for unranked terms and hedges. Journal of Automated Reasoning **52**(2), 155–190 (2014)
6. Leo, K., Tack, G.: Multi-pass high-level presolving. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 346–352. AAAI Press (2015), `http://ijcai.org/proceedings/2015`
7. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. Lecture Notes in Computer Science, vol. 10335, pp. 77–93. Springer (2017)
8. Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B.: A method for detecting symmetries in constraint models and its generalisation. Constraints **20**(2), 235–273 (2015)
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference. pp. 530–535. ACM (2001)
10. 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.) CP. LNCS, vol. 4741, pp. 529–543. Springer-Verlag (2007)
11. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = Lazy Clause Generation. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 4741, pp. 544–558. Springer (2007)
12. Plotkin, G.D.: A note on inductive generalization. Machine intelligence **5**(1), 153–163 (1970)
13. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and programming with Gecode (2016), `http://www.gecode.org`
14. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.: Why Cumulative Decomposition Is Not as Bad as It Sounds. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 746–761. Springer (2009)
15. Schutt, A., Stuckey, P.J., Verden, A.R.: Optimal carpet cutting. In: Lee, J. (ed.) Principles and Practice of Constraint Programming – CP 2011. pp. 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
16. Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M.: Learning from learning solvers. In: Rueher, M. (ed.) Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming. pp. 455–472. Lecture Notes in Computer Science, Springer (2016)
17. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine **35**(2), 55–60 (2014)