# Symmetries and Lazy Clause Generation

Geoffrey Chu[1], Maria Garcia de la Banda[2], Chris Mears[2], and Peter J. Stuckey[1]

[1] National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] Faculty of Information Technology,
Monash University, Australia
{cmears,mbanda}@infotech.monash.edu.au

**Abstract.** Lazy clause generation is a powerful approach to reducing search in constraint programming. This is achieved by recording sets of domain restrictions that previously lead to failure as new clausal propagators. Symmetry breaking approaches are also powerful methods for reducing search by recognizing that parts of the search tree are symmetric and do not need to be explored. In this paper we show how we can successfully combine symmetry breaking methods with lazy clause generation. Further, we show that the more precise nogoods generated by a lazy clause solver allow our combined approach to exploit redundancies that cannot be exploited via any previous symmetry breaking method, be it static or dynamic.

## 1 Introduction

Lazy clause generation [6] is a hybrid approach to constraint solving that combines features of finite domain propagation and Boolean satisfiability. Finite domain propagation is instrumented to record the reasons for each propagation step. This creates an implication graph like that built by a SAT solver, which may be used to create efficient nogoods that record the reasons for failure. These nogoods can be propagated efficiently using SAT unit propagation technology. The resulting hybrid system combines some of the advantages of finite domain constraint programming (high level model and programmable search) with some of the advantages of SAT solvers (reduced search by nogood creation, and effective autonomous search using variable activities). Thanks to this lazy clause generation provides state of the art solutions to a number of combinatorial optimization problems such as Resource Constrained Project Scheduling Problems [7].

Symmetry breaking methods aim at speeding up the execution by pruning parts of the search tree known to be symmetric to those explored. While static symmetry breaking methods achieve this by adding constraints to the original problem, dynamic symmetry breaking methods alter the search. As we will see later, combining static symmetry breaking with lazy clause generation is straightforward and quite successful. However, dynamic symmetry breaking can sometimes be more effective than static symmetry breaking. Thus, we are also

interested in combining lazy clause generation with dynamic symmetry breaking methods. While this combination is much more complex, it also allows us to exploit certain types of redundancies which were previously impossible to exploit via any other traditional static or dynamic symmetry breaking method.

As we will show in this paper, the key to the success of our combination resides in the fact that dynamic symmetry breaking methods can also be defined in terms of nogoods. In particular, they can be thought of as utilising symmetric versions of nogoods derived at each search node to prune off symmetric portions of the search space. Thus, both lazy clause generation and dynamic symmetry breaking use nogoods to prune the search space. The differences arise in the kind of nogoods used and in the way these nogoods are used. Traditional dynamic symmetry breaking methods such as SBDS [4] and SBDD [3, 1], use what we will call the *choice* nogood, i.e. the nogood formed by taking the entire set of current decision assignments. On the other hand, lazy clause solvers [6] use what is called the *first unique implication point* (1UIP) nogood (described in Section 3), which has been empirically found to be much stronger than choice nogoods in terms of pruning strength as clausal propagators. As our theoretical exploration will show, this difference in pruning strength carries over to dynamic symmetry breaking methods. Combining lazy clause generation and dynamic symmetry breaking allows us to take advantage of 1UIP nogoods (as lazy evaluation does) and of symmetric 1UIP nogoods (rather than of symmetric choice nogoods, as dynamic symmetry breaking does). This leads to strictly more pruning.

## 2 Finite Domain Propagation

Let $\equiv$ denote syntactic identity and $vars(O)$ denote the set of variables of object $O$. We use $\Rightarrow$ and $\Leftrightarrow$ to denote logical implication and logical equivalence, respectively.

A *constraint problem* $P$ is a tuple $(C, D)$, where $C$ is a set of constraints and $D$ is a *domain* which maps each variable $x \in vars(C)$ to a finite set of integers $D(x)$. The set $C$ is logically interpreted as the conjunction of its elements, while $D$ is interpreted as $\wedge_{x \in vars(C)} x \in D(x)$. A variable $x$ is said to be Boolean if $D(x) = [0, 1]$, where 0 represents *false* and 1 represents *true*.

An *equality literal* of $P \equiv (C, D)$ is of the form $x = d$, where $x \in vars(C)$ and $d \in D(x)$. A *valuation* $\theta$ *of* $P$ *over set of variables* $V \subseteq vars(C)$ is a set of equality literals of $P$ with exactly one literal per variable in $V$. It can be understood as a mapping of variables to values. The *projection* of valuation $\theta$ over a set of variables $U \subseteq vars(\theta)$ is the valuation $\theta_U = \{x = \theta(x) | x \in U\}$.

A constraint $c \in C$ can be considered a set of valuations $solns(c)$ over the variables $vars(c)$. Valuation $\theta$ *satisfies* constraint $c$ iff $vars(c) \subseteq vars(\theta)$ and $\theta_{vars(c)} \in c$. A *solution* of $P$ is a valuation over $vars(P)$ that satisfies every constraint in $C$. We let $solns(P)$ be the set of all its solutions. Problem $P$ is *satisfiable* if it has at least one solution and *unsatisfiable* otherwise.

An *inequality literal* for problem $P = (C, D)$ has the form $x \leq d$ or $x \geq d$ where $x \in vars(C)$ and $d \in D(x)$. A *disequality literal* for $x$ has the form $x \neq d$ where $d \in D(x)$. The equality, inequality and disequality literals of $P$, together with the special literal *false* representing failure, are denoted the *literals* of $P$. Literals represent the basic changes in domain that occur during propagation.

A constraint $c$ is implemented by a propagator $f_c$ which is a function from domains to domains that ensures that $c \wedge D \Leftrightarrow c \wedge f_c(D)$. We can record the new information obtained by running $f_c$ on domain $D$ as the set of literals which are newly implied: $new(f_c, D) = \{l \mid D \not\Rightarrow l \wedge f_c(D) \Rightarrow l\}$. We will assume that we remove from this set literals that are redundant. Note that if the propagator detects failure we assume $new(f_c, D) = \{false\}$.

*Example 1.* Consider the actions of propagator $f_c$ of constraint $c \equiv \sum_{i=1}^{5} x_i \leq 12$ on the domain $D(x_1) = \{1\}, D(x_2) = D(x_3) = D(x_4) = D(x_5) = [\,2\,..\,10\,]$. Now $D' = f_c(D)$ has $D(x_2) = D(x_3) = D(x_4) = D(x_5) = [\,2\,..\,5\,]$. Hence, as defined $new(f_c, D)$ includes $x_2 \geq 2, x_2 \leq 5, x_2 \leq 6, x_2 \leq 7, \ldots$. Since the second literal makes those following redundant, we assume they are not part of the result. □

Given a root constraint problem $P \equiv (C, D)$, constraint programming solves $P$ by a search process that first uses a constraint solver to determine whether $P$ can immediately be classified as satisfiable or unsatisfiable. We assume a propagation solver, denoted by $\mathsf{solv}$, which when applied to $P$ repeatedly applies propagators, updating the domain, until each returns an empty set of new literals. The final resulting domain $D'$ is such that $D' \Rightarrow D$ and $C \wedge D \Leftrightarrow C \wedge D'$. The solver detects unsatisfiability if any $D'(x) = \emptyset$ for some $x \in vars(C)$. We assume that if the solver returns a domain $D'$ where all variables are fixed then the solver has detected satisfiability of the problem and $D'$ is a solution. If the solver cannot immediately determine whether $P$ is satisfiable or unsatisfiable, the search splits $P$ into $n$ subproblems $P_i = (C \wedge c_i, D')$ where $C \wedge D' \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n))$ and iteratively searches for solutions to them.

The idea is for the search to drive towards subproblems that can be immediately detected by $\mathsf{solv}$ as being satisfiable or unsatisfiable. This solving process implicitly defines a *search tree* rooted by the original problem $P$ where each node represents a new (though perhaps logically equivalent) subproblem $P'$, which will be used as the node's label. In this paper we restrict ourselves to the case where each $c_i$ added by the search takes the form of a literal (referred to as a *decision* literal). While this is not a strong restriction, it does rule out some kinds of constraint programming search. We can identify any subproblem $P'$ appearing in the search tree for $P = (C, D)$ by the set of decision literals $c_1, \ldots, c_n$ taken to reach $P'$. We define $choices(P') = C'$ where $P' = (C \cup C', D')$.

## 3 Lazy Clause Generation

Lazy clause generation [6] is a hybrid of finite domain and SAT solving where each FD propagator is extended to be able to explain its propagations. An integer variable $x$ in problem $P = (C, D)$ with initial domain $D(x) = [\,l\,..\,u\,]$ is correlated to a set of Boolean variables $\{[\![x = d]\!] \mid l \leq d \leq u\} \cup \{[\![x \leq d]\!] \mid l \leq d < u\}$ that represent the domain changes possible for the variable (note that $[\![x \leq u]\!]$ is always true). Note that for variables $x$ with initial domain $D(x) = [\,0\,..\,1\,]$ we can represent them using the single Boolean variable $[\![x = 1]\!]$. In order to prevent meaningless assignments to these Boolean variables we add Boolean constraints

to the constraint problem that define the conditions that relate them.

$$\begin{aligned} [\![x \le d]\!] &\rightarrow [\![x \le d+1]\!], l \le d \le u-2 \\ [\![x \le d]\!] \wedge \neg[\![x \le d-1]\!] &\leftrightarrow [\![x = d]\!], \qquad l < d \le u-1 \\ [\![x \le l]\!] &\leftrightarrow [\![x = l]\!] \\ \neg[\![x \le u-1]\!] &\leftrightarrow [\![x = u]\!] \end{aligned}$$

Rather than directly using the Boolean variables attached to an integer variable we will use equality, inequality and disequality literals. Each equality, inequality and disequality literal can be considered as simply more explicit notation for a Boolean literal using the Boolean variables defined above:

$$\begin{array}{ll} x = d \equiv [\![x = d]\!] & x \le u \equiv \mathit{true} \\ x \ne d \equiv \neg[\![x = d]\!] & x \ge d \equiv \neg[\![x \le d-1]\!], l < d \\ x \le d \equiv [\![x \le d]\!], d < u & x \ge l \equiv \mathit{true}. \end{array}$$

For lazy clause generation, if a propagator $f_c$ implementing constraint $c$ infers a new literal (equality, inequality, or disequality) on the domain of one of its variables, or failure, it must explain this literal in terms of the Boolean representation of the variables involved in the propagator. An *explanation* for literal $l$ is $S \rightarrow l$ where $S$ is a set of literals. A correct explanation for $l$ by $f_c$ propagating on a problem with initial domain $D$, is an explanation $S \rightarrow l$ where $c \wedge S \wedge D \Rightarrow l$. Clearly, an explanation corresponds directly to a clause on the underlying Boolean representation. For example, the propagator for constraint $x \ne y$ may infer literal $y \ne 3$ given literal $x = 3$. This might be explained as $\{x = 3\} \rightarrow y \ne 3$ corresponding to the clause $[\![x = 3]\!] \rightarrow \neg[\![y = 3]\!]$.

In a lazy clause generation solver each new literal $l$ inferred by a propagator $f_c$ is recorded in a stack in the order of generation. Furthermore, the propagator returns an explanation for $l$ that is attached to $l$. The *implication graph* is thus a stack of literals each with an attached explanation, or marked as a decision literal. We define the *decision level* for any literal as the number of decision literals pushed in before it in the stack.

*Example 2.* Consider the following constraint problem $P = (C, D)$ where $C \equiv \{\sum_{i=1}^{5} x_i \le 12,\ \mathit{alldiff}([x_1, x_2, x_3, x_4, x_5])\}$ and $D(x_i) = [1..8], 1 \le i \le 5$. If the search chooses $x_1 = 1$ we arrive at subproblem $P_1 = (C \cup \{x_1 = 1\}, D)$. Then the *alldiff* constraint determines that $x_2 \ne 1$ from set of literals $\{x_1 = 1\}$ (i.e., with explanation $\{x_1 = 1\} \rightarrow x_2 \ne 1$), and similarly for $x_3, x_4$ and $x_5$. This builds the second column of the implication graph in Figure 1. Then, the domain constraints for $x_2$ determine that $x_2 \ge 2$ from $\{x_2 \ne 1\}$ and similarly for the domain constraints of $x_3, x_4$, and $x_5$, building the third column. The sum constraint determines that the upper bound of each of $x_2$, $x_3$, $x_4$ and $x_5$ is 5 from the lower bounds in the third column, thus building the fourth column. The new domain is $D'(x_1) = \{1\}$, $D'(x_i) = [2..5], 2 \le i \le 5$. If the search now chooses $x_2 = 2$, we arrive at subproblem $P_2 = (C \cup \{x_1 = 1, x_2 = 2\}, D')$. Then the *alldiff* constraint determines $x_3 \ne 2$, $x_4 \ne 2$, and $x_5 \ne 2$ (the 6th column) from $\{x_2 = 2\}$. The domain constraints determine that $x_3 \ge 3$ from $\{x_3 \ge 2, x_3 \ne 2\}$, similarly for $x_4$ and $x_5$. The sum constraint determines that $x_4 \le 3$ from $\{x_2 = 2, x_3 \ge 3, x_5 \ge 3\}$, similarly for $x_5 \le 3$. Then, the domain

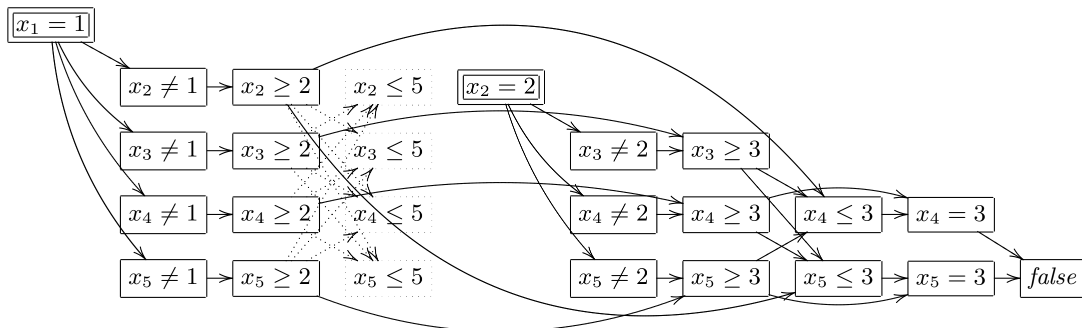


**Fig. 1.** Implication graph of propagation. Decision literals are double boxed.

constraints determine $x_4 = 3$ from $\{x_4 \geq 3, x_4 \leq 3\}$, similarly for $x_5 = 3$ and finally the *alldiff* constraint determines unsatisfiability of $x_4 = 3$ and $x_5 = 3$. □

A *nogood* $N$ is set of literals. A correct nogood $N$ from problem $P = (C, D)$ is one where $C \wedge D \Rightarrow \neg \wedge_{l \in N} l$, that is, in all solutions of $P$ the conjunction of the literals in $N$ is false. Once we have an implication graph we can use it determine a correct nogood that explains each failure. The usual approach to building a nogood is to use the implication graph to eliminate literals starting from original nogood $N$ from the explanation of failure $N \rightarrow$ *false*, until only one literal at the the current decision level remains. This is the 1UIP (First Unique Implication Point) nogood. The search then records this nogood as a clausal propagator and backtracks to the decision level of the second latest literal in the nogood, where it applies the newly derived nogood propagator.

*Example 3.* Continuing from Example 2 using the implication graph in Figure 1, we start with the explanation of failure $\{x_4 = 3, x_5 = 3\} \rightarrow$ *false* which gives us the initial nogood $\{x_4 = 3, x_5 = 3\}$. Since both literals were determined at the current decision level, we replace the last one $x_5 = 3$ by the antecedents in its explanation $\{x_5 \geq 3, x_5 \leq 3\} \rightarrow x_5 = 3$ to obtain $\{x_5 \geq 3, x_5 \leq 3, x_4 = 3\}$. We keep removing the last literal with the current decision level until only one literal remains at the current decision level. The resulting 1UIP nogood is $\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_5 \geq 2, x_2 = 2\}$ which can be simplified to $\{x_3 \geq 2, x_4 \geq 2, x_5 \geq 2, x_2 = 2\}$, since the last literal implies the first.

On backtracking to undo the choice $x_2 = 2$, the search arrives at subproblem $P_1$ and immediately determines that $x_2 \neq 2$ using the new nogood. The important point is that if the search ever reaches a state where $\{x_3 \geq 2, x_4 \geq 2, x_5 \geq 2\}$ hold we will make the same inference, or indeed if it reaches a point where $\{x_2 = 2, x_3 \geq 2, x_4 \geq 2\}$ we will infer that $x_5 < 2$. □

In general, we restrict ourselves to creating nogoods which are *asserting*, that is, there should be only a single literal $l$ in the nogood with the latest decision level. This allows us, upon backtracking to the decision level of the second latest literal, to *assert* $\neg l$ since the remaining literals are true. Another

possible asserting nogood generation approach is the so called *decision* nogood, where we start from the original explanation of failure and keep eliminating all literals which are not decision literals (that is, have an explanation). This builds much weaker nogoods in general than 1UIP nogoods.

*Example 4.* Nogood $\{x_2 \geq 2, x_3 \geq 3, x_4 \geq 3, x_5 \geq 3\}$ is correct for Example 2 but is not asserting since the last 3 literals belong to the latest decision level. The decision nogood for Example 2 is $\{x_1 = 1, x_2 = 2\}$. □

## 4 Symmetries and Nogoods

A *symmetry* of constraint problem $P = (C, D)$ is a bijection $\rho$ on the equality literals of $P$ such that, for each valuation $\theta$ of $P$, $\rho(\theta) = \{\rho(l) \mid l \in \theta\}$ is a solution of $P$ iff $\theta$ is a solution of $P$. Variable symmetries, value symmetries and variable-value symmetries are all particular cases of symmetries.

*Example 5.* A variable symmetry $\rho$ swapping variables $x_1$ and $x_2$ is defined as $\rho(x_1 = d) \equiv (x_2 = d)$, $\rho(x_2 = d) \equiv (x_1 = d)$, and $\rho(v = d) \equiv (v = d), v \notin \{x_1, x_2\}$ for all values $d$. We denote it $\ll x_1 \gg \leftrightarrows \ll x_2 \gg$. A value symmetry $\rho$ swapping value 1 for 3 and 2 for 4 is defined by $\rho(v = 1) \equiv (v = 3)$, $\rho(v = 2) \equiv (v = 4)$, $\rho(v = 3) \equiv (v = 1)$, $\rho(v = 4) \equiv (v = 2)$, $\rho(v = d) \equiv (v = d), d \notin \{1, 2, 3, 4\}$ for all variables $v \in \mathit{vars}(P)$. We denote it $\ll 1, 2 \gg \leftrightarrows \ll 3, 4 \gg$. □

Static Symmetry Breaking effectively reduces the search required to find the first, all or the best solution to a constraint problem by adding constraints that remove symmetric solutions. In particular, lexicographical constraints have been used to statically eliminate symmetries (see e.g.[2]) with excellent results. This is good news since static symmetry breaking is obviously compatible with lazy clause generation: we only require the new symmetry breaking constraints to have explaining propagators, which are used just like other propagators. However, static symmetry breaking is not always the best option. If we have multiple symmetries, care must be taken so that the static symmetry breaking constraints for each symmetry do not interact badly. Also, static symmetry breaking constraints may interact badly with a given search strategy, making the search take even longer to find a solution. Hence, we are also interested in dynamic symmetry breaking methods.

Dynamic Symmetry Breaking techniques can be interpreted as pruning symmetric portions of the search space by propagating symmetric versions of nogoods. Consider a search strategy where only equality literals are used to split search, as it is usual for symmetry papers. Then if subproblem $P'$ fails $\mathit{choices}(P')$ is a correct nogood of $P$. Let us denote this as the *choice nogood*. Since a generated nogood $N$ is a globally true statement, it holds at any point during the search and, hence, any symmetric version of $N$ is also a correct nogood. Note that the symmetric version, $\rho(N)$, of a nogood $N$ consisting of only equality literals is easy to define: $\rho(N) = \{\rho(l) \mid l \in N\}$.

*Example 6.* In problem $P$ of Example 2 the variables $\{x_1, x_2, x_3, x_4, x_5\}$ are indistinguishable (i.e., any two can be swapped). Since the subproblem $P'$ with

$choices(P') = \{x_1 = 1, x_2 = 2\}$ fails, we have that $\{x_1 = 1, x_2 = 2\}$ is a correct nogood for $P$. Clearly, any symmetric version, such as $\{x_2 = 1, x_1 = 2\}$ or $\{x_3 = 1, x_5 = 2\}$, is also a correct nogood. □

Such nogoods can be used to prune search in two main ways. Symmetry breaking by dominance detection (SBDD) [3, 1] keeps a store $\mathbf{N}$ of the non-subsumed choice nogoods derived during search so far. For each subproblem $P'$, it checks whether there exists $N \in \mathbf{N}$ and symmetry $\rho$, such that $choices(P') \Rightarrow \rho(N)$. If such a pair exists it can immediately fail subproblem $P'$. Symmetry breaking during search (SBDS) [4] works as follows. Whenever a subproblem $P'$ with $choices(P') = \{d_1, d_2, \ldots, d_n, d_{n+1}\}$ fails, SBDS backtracks to the parent suproblem $P''$ in level $n$ and, for each symmetry $\rho$, it locally posts in $P''$ the conditional constraint $(\rho(d_1) \wedge \ldots \wedge \rho(d_n)) \rightarrow \neg\rho(d_{n+1})$. Note that these constraints will only propagate when reaching a subproblem $P'''$ such that $C \cup choices(P''')$ entails the left hand side of the constraint. This will never happen if the symmetry is *broken*, i.e., if $\exists d_i$ s.t. $\neg\rho(d_i)$ is entailed, and that is why SBDS ignores any symmetry $\rho$ which is known to be broken at $P''$. Still, SBDS can post too many local constraints when the number of symmetries is high. Thus, some incomplete methods ([5] and the shortcut method in [4]) post only those constraints that are known to immediately propagate.

We decided to integrate SBDS, rather than SBDD, with our lazy clause generation since SBDS is much closer to the lazy clause generation approach: they both compute and post nogoods. The main differences being that SBDS only computes decision nogoods and posts symmetric versions of these nogoods.

## 5 Symmetries and Lazy Clause Generation

### 5.1 SBDS-choice

We can naively add SBDS to a lazy clause solver by simply using symmetric versions of the choice nogood at each node to prune off symmetric branches. Hence, we just reimplement standard SBDS in the lazy clause generation solver, but still gain the advantage of reduced search through the lazy clause generation nogoods.

### 5.2 SBDS-1UIP

Adapting SBDS to use 1UIP nogoods is simple: every time a 1UIP nogood $\{l_1, \ldots, l_n\} \rightarrow l_{n+1}$ is inferred for subproblem $P'$, upon backtracking to parent $P''$ and for each symmetry $\rho$, we post the symmetric nogood $\{\rho(l_1), \ldots, \rho(l_n)\} \rightarrow \neg\rho(l_{n+1})$, ignoring those $\rho$ that are known to be broken at $P''$. We can check this last condition during the construction of the symmetric nogood, as we produce the literals $\rho(l_1), \ldots, \rho(l_n)$ one at a time. If at any point, one of $\rho(l_i)$ is false in $P''$, we can immediately abort and move on to the next symmetry.

In contrast to SBDS-choice, in SBDS-1UIP we have to post the symmetric nogoods as global rather than local constraints. This is because in SBDS-choice, when you backtrack from parent $P''$ to grandparent $P'''$, the choice nogood at $P'''$ subsumes that at $P''$ and, therefore, SBDS-choice will always post a set of

symmetric nogoods that subsumes the symmetric nogoods posted below that point. In contrast, there is no guarantee that the 1UIP nogood at $P'''$ subsumes the one at $P''$ (and in general it doesn't).

*Example 7.* Consider the problem of Example 2. On backtracking to $P_1$ we infer the nogood $\{x_3 \geq 2, x_4 \geq 2, x_5 \geq 2\} \rightarrow x_2 \neq 2$. With this we not only infer $x_2 \neq 2$ but also the symmetric inferences $x_3 \neq 2$ (from $\{x_2 \geq 2, x_4 \geq 2, x_5 \geq 2\}$), $x_4 \neq 2$ and $x_5 \neq 2$. At this point, a domain consistent *alldiff* will determine unsatisfiability, and generate the nogood $\emptyset \rightarrow x_1 \neq 1$, which does not imply the previously generated nogood. □

We show that SBDS-1UIP exploits strictly more symmetries than SBDS-choice if the asserting literals in the nogoods are the same, and propagation has the following property:

**Definition 1.** *A set of propagators for problem $P$ has* global symmetric monotonicity *iff, for any explanation $\{d_1, \ldots, d_n\} \rightarrow l$ produced and any symmetry $\rho$ of $P$, whenever $\rho(d_1), \ldots, \rho(d_n)$ are entailed, then $\rho(l)$ must also be entailed.* □

A sufficient condition for global symmetric monotonicity is the following: all propagators are monotonic, and all symmetries are propagator symmetric (propagators map to propagators under the symmetry). The proof of this is straightforward and we omit it for lack of space. Global symmetric monotonicity is therefore very common, as most propagators are monotonic, and the vast majority of symmetries that are usually exploited are propagator symmetric.

**Theorem 1.** *Suppose global symmetric monotonicity holds, and we derive the choice nogood $\{d_1, \ldots, d_n\} \rightarrow \neg d_{n+1}$, and the 1UIP nogood $\{l_1, \ldots, l_m\} \rightarrow \neg d_{n+1}$ from the same conflict. If the nogood $\{\rho(d_1), \ldots, \rho(d_n)\} \rightarrow \neg\rho(d_{n+1})$ propagates then so does $\{\rho(l_1), \ldots, \rho(l_m)\} \rightarrow \neg\rho(d_{n+1})$.*

*Proof.* Suppose the symmetric version of the choice nogood can propagate for domain $D'$. Then $\rho(d_1), \ldots, \rho(d_n)$ must all be entailed. From the implication graph from which we derived the 1UIP nogood, we know that $d_1 \wedge \ldots \wedge d_n \Rightarrow l_i$ for any $i$. Since global symmetric monotonicity holds and $\rho(d_1), \ldots, \rho(d_n)$ are entailed, we know that $\rho(l_i)$ must also be entailed for any $i$. This means that the symmetric version of the 1UIP nogood can also propagate. □

The theorem shows that the symmetric 1UIP nogood subsumes the symmetric choice nogood, since it will always produce any implication that the symmetric choice nogood can, but not vice versa. This means that SBDS-1UIP can exploit strictly more symmetry than SBDS-choice. This result is valid for both complete SBDS, as well as for incomplete SBDS methods which only post nogoods that will immediately produce an implication.

### 5.3 Beyond complete methods?

The previous result is somewhat surprising considering that SBDS-choice is a "complete" symmetry breaking method, which guarantees that once we have examined a certain partial assignment, we will never examine any symmetric

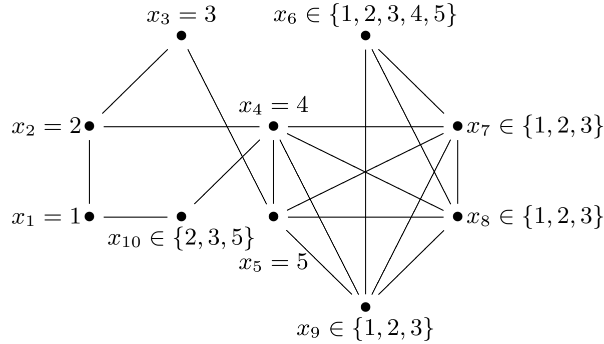


**Fig. 2.** A graph colouring problem where we can exploit additional symmetries

version of it. However, this does not actually mean that we have exploited all possibly redundancies arising from symmetry. Roughly speaking, SBDS can only exploit symmetries on the already labelled parts of the problem. It is incapable of exploiting symmetries in the unlabelled parts of the problem.

*Example 8.* Consider the graph colouring problem shown in Figure 2, where we are trying to colour the nodes with at most 5 colours (all of which are interchangeable). After making the decisions $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5$, we have domains as shown in Figure 2. Suppose we label $x_6 = 1$ next. Then propagation gives $x_7 \in \{2, 3\}, x_8 \in \{2, 3\}, x_9 \in \{2, 3\}$. Now, suppose we try $x_7 = 2$. This forces $x_8 = 3, x_9 = 3$, which conflicts. The 1UIP nogood from this conflict is $\{x_8 \neq 1, x_8 \neq 4, x_8 \neq 5, x_9 \neq 1, x_9 \neq 4, x_9 \neq 5\} \rightarrow x_7 \neq 2$. After propagating this nogood, we have $x_7 = 3$, which after further propagation, once again conflicts. At this point, we backtrack to before $x_6$ is labelled and derive the nogood $\{x_7 \neq 4, x_7 \neq 5, x_8 \neq 4, x_8 \neq 5, x_9 \neq 4, x_9 \neq 5\} \rightarrow x_6 \neq 1$.

Now, let's examine what SBDS-1UIP can do at this point. It is clear that if we apply the value symmetries $\ll 1 \gg \leftrightarrows \ll 2 \gg$ or $\ll 1 \gg \leftrightarrows \ll 3 \gg$ to this nogood, the LHS remains unchanged while the RHS changes. Therefore, we can post these two symmetric nogoods and immediately get the inferences $x_6 \neq 2$ and $x_6 \neq 3$. On the other hand, SBDS-choice can't do anything. The choice nogood is $\{x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5\} \rightarrow x_6 \neq 1$, and it is easy to see that no matter which value symmetry we use on it, the LHS will have a set of literals incompatible with the current set of decisions and thus cannot imply the RHS. □

The kind of redundancy we exploit here certainly arises from symmetry. However, it is extremely difficult to exploit. Roughly speaking, we can say that we are exploiting the symmetry that exists in the sub-component of a subproblem which is the actual cause of failure. In this case, they are the variables $x_6, x_7, x_8, x_9$, their current domains in the subproblem, and the constraints linking them. Even conditional symmetry breaking constraints are powerless to exploit such symmetries, as the subproblem shown in Figure 2 does not have the value symmetries $\ll 1 \gg \leftrightarrows \ll 2 \gg$ or $\ll 1 \gg \leftrightarrows \ll 3 \gg$ due to the existence of $x_{10}$. It is only because

a lazy clause solver gives us such precise information about which variables are involved in failures that we can exploit this kind of redundancy. Although the above example might seem somewhat contrived, we show in our experiments in Section 7 that these kinds of redundancies do occur in practice and can be exploited for more speedup.

# 6 Symmetries on 1UIP nogoods

SBDS-1UIP is much more powerful than SBDS-choice, however, having to manipulate 1UIP nogoods raises a whole host of other problems. In particular, unlike the choice nogoods which usually only involve equality literals on search variables, 1UIP nogoods can contain virtually any literal in the problem, i.e. they may include disequality literals, inequality literals, and also literals involving intermediate variables. The last is a rather serious issue as symmetries are often defined in terms of the output or search variables, and may not properly describe how intermediate variables map to each other. We now examine each of these issues in more detail.

## 6.1 Disequality and Inequality Literals

One of the strengths of lazy clause generation is the use of both equality literals and inequality literals in explanations and nogoods. This makes many explanations much shorter and is effectively essential for explaining bounds propagation.

Extending a literal symmetry $\rho$ to disequality literals is straightforward: if $\rho(x = d) \equiv x' = d'$, then $\rho(x \neq d) \equiv x' \neq d'$. Extending a literal symmetry $\rho$ to map inequality literals is harder. Given a nogood $N$ which may involve equality, inequality and disequality literals and a *variable* symmetry $\sigma$ it is easy to generate $\sigma(N)$ by simply applying the variable renaming $\sigma$ to $N$.

*Example 9.* Consider the problem of Example 2. This problem has variable interchangeability symmetries since each of $\{x_1, x_2, x_3, x_4, x_5\}$ are indistinguishable. Hence, any variable renaming of the generated nogood $\{x_3 \geq 2, x_4 \geq 2, x_5 \geq 2, x_2 = 2\}$ is valid, e.g. $\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_5 = 2\}$ or $\{x_1 \geq 2, x_3 \geq 2, x_5 \geq 2, x_4 = 2\}$ □

However, it's not so straightforward for value symmetries and variable-value symmetries. For such symmetries, inequality literals do not map simply to other literals. To apply such a symmetry to a nogood then, we first need to transform the nogood into an equivalent nogood involving only equality and disequality literals. After this we can apply the symmetry as usual.

Assume that in $P = (D, C)$ that $D(x) = [\, l \,..\, u \,]$ then the transformation $\mathsf{eq}$ on $x$ literals is defined as

$$\mathsf{eq}(x = d) \equiv \{x = d\} \qquad \mathsf{eq}(x \neq d) \equiv \{x \neq d\}$$
$$\mathsf{eq}(x \leq d) \equiv \{x \neq d' \mid d \leq d' < u\} \qquad \mathsf{eq}(x \geq d) \equiv \{x \neq d' \mid l < d' \leq d\}$$

We can extend this to a nogood $N$: $\mathsf{eq}(N) = \cup_{l \in N} \mathsf{eq}(l)$. We can then define the symmetric version of a nogood $N$ for any symmetry $\rho$ defined as a bijection on

equality literals as $\rho(N) = \{\rho(l) \mid l \in \texttt{eq}(N)\}$ Of course while this transformation to equality and disequality literals is theoretically fine, in practice it may create very unwieldy nogoods.

We can, in effect, implement this transformation by slightly modifying the nogood learning process. We will require that no inequality literals appear in the nogood, hence we must continue to explain them until none remain. As long as all decisions are either equality or disequalities this will still result in asserting nogoods always being discovered.

*Example 10.* Revisiting the explanation process of Example 3, the nogood discovered includes inequality literals, so rather than stopping the explanation process at this point we continue. The current nogood is $\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\}$, we explain each of the inequalities using the implication graph to arrive at $\{x_2 \neq 1, x_3 \neq 1, x_4 \neq 1, x_2 = 2\}$ which can again be simplified to $\{x_3 \neq 1, x_4 \neq 1, x_2 = 2\}$ and which does involve inequality literals. □

### 6.2 Intermediate variables

An important problem for combining dynamic symmetry breaking and lazy clause generation is the fact that intermediate variables may be introduced in the course of converting a high level model to the low level variables and constraints implemented by the solver. For example, a high level model written in the modeling language MiniZinc is first flattened into primitive constraints, with intermediate variables introduced as necessary, and then given to a solver, which may then introduce its own variables, e.g. in global propagators implemented by decomposition. 1UIP nogoods often contain literals from such intermediate variables. However, if the symmetry declaration was made only in the high level model, it may not specify how literals on such introduced intermediate variables map to each other. Thus it is necessary to consider how symmetries can be extended to include the literals on intermediate variables.

Intermediate variables are sometimes idempotent under the symmetries, that is for each symmetry $\rho$ of $P$, we can extend $\rho$ to $\rho'$ where $\rho'(l) = \rho(l), \mathit{vars}(l) \subseteq \mathit{vars}(C)$ and $\rho'(l) = l$ otherwise. The extended $\rho'$ is a symmetry of the problem with intermediate variables. We can imagine automating the proof of idempotence of intermediate variables under symmetries.

*Example 11.* Consider a model for concert hall scheduling The problem has a value interchangeability between all values $[\,1\,..\,k\,]$ for the $k$ identical concert halls. The model includes the constraint

```
constraint forall (i, j in Offers where i < j /\ o[i,j])
    (x[i] = k+1 \/ x[j] = k+1 \/ x[i] != x[j]);
```

which requires that for two overlapping concerts $i$ and $j$ (input data $o[i,j]$ is *true*) either $i$ is not scheduled (represented as the hall used $x[i]$ is $k+1$), $j$ is not scheduled, or the halls used are different. But this constraint is implemented, by reification, as something equivalent to

```
array[Offers]  of var bool: unscheduled;
array[Offers,Offers] of var bool: different;
```

```
constraint forall(i in Offers)(unscheduled[i] = (x[i] = k+1));
constraint forall (i, j in Offers where i < j /\ o[i,j])(
    different[i,j] = (x[i] != x[j]) /\
    (unscheduled[i] \/ unscheduled[j] \/ different[i,j]));
```

since the clausal propagator works on Boolean variables, and hence we need to reify the subexpressions. Each introduced variable $unscheduled[i]$ and $different[i, j]$ is idempotent under the value symmetries. Hence, for any symmetry $\rho$ on the original variables we can extend it trivially. □

Sometimes we need to extend our symmetry declarations to take into account the intermediate variables.

*Example 12.* In the graceful graph problem each node is labelled by an number from 0 to the number of edges. The difference between each edges node labels must be different. This is encoded as

```
constraint alldifferent([ abs(m[o[i]] - m[d[i]]) | i in Edges]);
```

where $m$ is the labelling on nodes, and $o[i], d[i]$ are the origin and destination of edge $i$. This constraint is implemented by flattening as something equivalent to

```
array[Edges] of var int: diff;
constraint forall(i in Edges)(diff[i] == m[o[i]]  - m[d[i]]);
srray[Edges] of var int: adiff;
constraint forall(i in Edges)(adiff[i] == abs(diff[i]));
constraint alldifferent(adiff);
```

The graceful graph problem can have symmetries arising from symmetries in the underlying graph. Suppose the underlying graph has 3 nodes and 2 edges (1,2) and (3,2) numbered 1 and 2. There is a symmetry between the two edges captures by the row interchangeability $\rho = \ll m[1], m[2] \gg \leftrightarrows \ll m[3], m[2] \gg$ which indicates we can swap the edges.

Once we consider the intermediate variables, we need to extend this symmetry to $\ll m[1], m[2], \mathit{diff}[1], \mathit{adiff}[1] \gg \leftrightarrows \ll m[3], m[2], \mathit{diff}[2], \mathit{adiff}[2] \gg$ thus interchanging all information on about the edges simultaneously. □

Sometimes it is not easy to see how to extend symmetries to all intermediate variables, and indeed quite often intermediate variables are introduced far below the modelling level. In order to handle these cases we modify learning as follows.

We extend the model to explicitly mark which literals are allowed to appear in nogoods. Then we modify the learning process to always explain any literals that are not marked. There is a requirement that all literals generated by search are allowed to appear in nogoods. This ensures that the process always terminates and always generates an asserting nogood.

*Example 13.* In order to tell the solver that the concert hall scheduling model that it can use only equality and disequality literals for the $x$ variables as well as the the intermediate variables in nogoods we annotate the declarations:

```
array[Offers] of var 1..k+1:x :: symmetric_nogoods_eq;
array[Offers] of var bool: unscheduled :: symmetric_nogoods;
array[Offers,Offers] of var bool: different :: symmetric_nogoods;
```

The declarations ensure that any other literal will never appear in a nogood to which we apply symmetry. Note that his means that bounds literals $x[i] \leq d$ will be replaced by inequality literals. □

## 7 Experiments

We now provide experimental evidence for the claims we made in the earlier parts of the paper. The two problem we will examine are the Concert Hall Scheduling problem and the Graph Colouring problem. We take the benchmarks used by [5].The benchmarks are available at `http://www.cmears.id.au/symmetry/symcache.tar.gz`.

We implemented SBDS in Chuffed, which is a state of the art lazy clause solver. We run Chuffed with three different versions of SBDS. The first version is choice, where we use symmetric versions of choice nogoods. The second is 1UIP, where we use symmetric versions of 1UIP nogoods. The third version we call crippled, where we use symmetric versions of 1UIP nogoods, but only those nogoods derived from symmetries where choice could also exploit the symmetry. We compare against Chuffed with no symmetry breaking (none) and with statically added symmetry breaking constraints (static). Finally, we compare against an implementation of SBDS in [5], which is called Lightweight Dynamic Symmetry Breaking (LDSB). LDSB is implemented on the Eclipse constraint programming platform and was the fastest implementations of dynamic symmetry breaking on the two problems we examine, beating GAP-SBDS and GAP-SBDD by significant margins.

All versions of Chuffed are run on Xeon Pro 2.4GHz processors. The results for LDSB were run on an Core i7 920 2.67 GHz processor. We group the instances by size, so that the times displayed are the average run times for the instances of each size. A timeout of 600 seconds was used. Instances which timeout are counted as 600 seconds.

The results are shown in Tables 1 and 2. Eclipse LDSB fails to solve many instances before timeout, and choice fails to solve a few instances. 1UIP, crippled and static all solve every instance in the benchmarks. In fact, this set of instances, which is of an appropriate size for normal CP solvers, is a bit too easy for lazy clause solvers such as Chuffed, as is apparent from the run times.

Comparison between choice and 1UIP shows that SBDS-1UIP is superior to SBDS-choice. Comparison between crippled and 1UIP shows that the additional symmetries that we can only exploit with SBDS-1UIP indeed gives us reduced search and additional speedup. Comparison with static shows that dynamic symmetry breaking can be superior to static symmetry breaking on appropriate problems. The comparison with LDSB shows that lazy clause solvers can be much faster than normal CP solvers, and that they retain this advantage when integrated with symmetry breaking methods. It also show by proxy that SBDS-1UIP is superior to GAP-SBDS or GAP-SBDD on these problems.

The total speed difference between 1UIP and LDSB is up to 2 orders of magnitude for the Concert Hall problems and up to 4 orders of magnitude for the Graph Colouring problems. Most of this speedup can be explained by the dramatic reduction in search space, which is apparent from the node counts in the

**Table 1.** Comparison of three SBDS implementations in Chuffed, static symmetry breaking in Chuffed, and LDSB in Eclipse, on the Concert Hall Scheduling problem

| Size | none | | 1UIP | | crippled | | choice | | static | | LDSB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Nodes |
| 20 | 259.8 | 686018 | 0.04 | 84 | 0.05 | 130 | 0.07 | 350 | 0.05 | 134 | 0.29 | 3283 |
| 22 | 381.5 | 749462 | 0.07 | 181 | 0.08 | 299 | 0.17 | 1207 | 0.07 | 183 | 0.73 | 7786 |
| 24 | 576.9 | 1438509 | 0.10 | 275 | 0.11 | 316 | 0.78 | 3426 | 0.15 | 486 | 2.70 | 12611 |
| 26 | 483.4 | 1189930 | 0.10 | 282 | 0.19 | 677 | 2.26 | 5605 | 0.25 | 685 | 2.71 | 12724 |
| 28 | 530.7 | 1282797 | 0.68 | 1611 | 1.12 | 2613 | 3.64 | 10530 | 0.42 | 1041 | 9.94 | 57284 |
| 30 | 581.3 | 1251980 | 0.27 | 761 | 0.53 | 2042 | 19.52 | 48474 | 0.52 | 2300 | 121.50 | 722668 |
| 32 | – | – | 0.40 | 1522 | 1.01 | 4845 | 21.48 | 65157 | 1.31 | 5712 | 97.90 | 641071 |
| 34 | – | – | 1.10 | 2636 | 3.22 | 8761 | 19.86 | 48837 | 1.60 | 4406 | 72.73 | 425718 |
| 36 | – | – | 1.40 | 3156 | 5.02 | 13606 | 59.70 | 131142 | 2.37 | 5707 | 171.14 | 938439 |
| 38 | – | – | 1.91 | 5053 | 12.56 | 26556 | 82.77 | 178170 | 3.51 | 10518 | 268.05 | 1211086 |
| 40 | – | – | 2.96 | 6648 | 10.27 | 27028 | 102.1 | 219454 | 6.40 | 18169 | 240.84 | 1220934 |

**Table 2.** Comparison of three SBDS implementations in Chuffed, static symmetry breaking in Chuffed, and LDSB in Eclipse, on the Graph Colouring problems

Uniform

| Size | none | | 1UIP | | crippled | | choice | | static | | LDSB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Nodes |
| 30 | 140.7 | 282974 | 0.00 | 14 | 0.06 | 474 | 0.26 | 3049 | 0.02 | 277 | 19.63 | 56577 |
| 32 | 211.4 | 390392 | 0.00 | 17 | 0.00 | 146 | 0.24 | 3677 | 0.00 | 84 | 14.11 | 27178 |
| 34 | 213.9 | 272772 | 0.00 | 25 | 0.29 | 1182 | 3.53 | 11975 | 0.03 | 433 | 22.06 | 30127 |
| 36 | – | – | 0.00 | 36 | 0.04 | 467 | 6.91 | 23842 | 0.01 | 200 | 35.66 | 85505 |
| 38 | – | – | 0.00 | 55 | 0.04 | 516 | 23.55 | 69480 | 0.03 | 526 | 51.18 | 107574 |
| 40 | – | – | 0.00 | 83 | 0.31 | 1879 | 21.07 | 78918 | 0.06 | 878 | 84.16 | 185707 |

Biased

| Size | none | | 1UIP | | crippled | | choice | | static | | LDSB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Fails | Time | Nodes |
| 20 | 13.25 | 39551 | 0.00 | 27 | 0.00 | 32 | 0.01 | 639 | 0.00 | 29 | 0.72 | 1376 |
| 22 | 11.53 | 63984 | 0.00 | 25 | 0.00 | 34 | 0.02 | 727 | 0.00 | 25 | 0.16 | 538 |
| 24 | 66.60 | 154409 | 0.00 | 35 | 0.00 | 47 | 0.07 | 1992 | 0.00 | 32 | 1.91 | 2114 |
| 26 | 74.77 | 277290 | 0.00 | 55 | 0.00 | 93 | 0.12 | 3385 | 0.00 | 104 | 9.56 | 34210 |
| 28 | 130.5 | 280649 | 0.00 | 62 | 0.00 | 84 | 0.58 | 6402 | 0.00 | 103 | 9.14 | 37738 |
| 30 | 267.6 | 480195 | 0.00 | 101 | 0.01 | 239 | 10.48 | 43835 | 0.01 | 359 | 57.18 | 215932 |
| 32 | 331.7 | 600772 | 0.01 | 232 | 0.24 | 1597 | 9.98 | 44216 | 0.16 | 1864 | 110.16 | 288707 |
| 34 | 219.9 | 387213 | 0.20 | 806 | 0.40 | 1946 | 10.26 | 47470 | 0.45 | 1730 | 64.14 | 165943 |
| 36 | – | – | 0.01 | 317 | 0.04 | 857 | 27.39 | 113252 | 0.80 | 3226 | 152.62 | 327472 |
| 38 | – | – | 0.10 | 798 | 1.01 | 5569 | 31.63 | 138787 | 4.12 | 9413 | 193.40 | 508164 |
| 40 | – | – | 0.02 | 410 | 0.36 | 2660 | 24.68 | 91847 | 0.12 | 2133 | 196.02 | 476486 |

results table. The redundancies exploited by lazy clause solvers are different from those redundancies caused by symmetries, and it is very clear here that by exploiting both at the same time with SBDS-1UIP, we get much higher speedups than possible with either of them separately. It should also be noted that Chuffed with static symmetry breaking constraints is also reasonably fast. While symme-

try breaking constraints cannot exploit the extra redundancies that SBDS-1UIP can, it does have very low overhead and integrates well with lazy clause.

## 8 Conclusion

In this paper we have examined dynamic symmetry breaking methods, and understood them as manipulating the choice nogoods created by normal depth first search. We show how we can extend these approaches to make use of the better nogoods generated by lazy clause solvers. This extension introduces a number of new issues, such as how to deal with disequality and inequality literals, and literals from intermediate variables. We have built a prototype implementation combining SBDS with lazy clause generation, which we call SBDS-1UIP. The resulting system can exploit types of redundancies previously impossible to exploit, and can outperform LDSB by several orders of magnitudes on some problems.

**Acknowledgements**. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

## References

1. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference*, pages 93–107, 2001.
2. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference*, pages 462–476, 2002.
3. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 77–92, 2001.
4. I. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *14th European Conference on Artificial Intelligence*, pages 599–603, 2000.
5. C. Mears. *Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming*. PhD thesis, Clayton School of Information Technology, Monash University, 2010.
6. O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
7. A. Schutt, T. Feydy, P.J. Stuckey, and M. Wallace. Why cumulative decomposition is not as bad as it sounds. In I. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, pages 746–761. Springer-Verlag, 2009.