Implementing Logical Connectives in Constraint Programming

Christopher Jefferson†, Neil C.A. Moore†, Peter Nightingale†, Karen E. Petrie*

* School of Computing, University of Dundee, Dundee DD1 4HN, UK
email: kpetrie@computing.dundee.ac.uk
† School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK
email: caj@cs.st-andrews.ac.uk, ncam@cs.st-andrews.ac.uk, pn@cs.st-andrews.ac.uk

Abstract

Combining constraints using logical connectives such as disjunction is ubiquitous in constraint programming, because it adds considerable expressive power to a constraint language. We explore the solver architecture needed to propagate such combinations of constraints efficiently. In particular we describe two new features named *satisfying sets* and *constraint trees*. We also make use of *movable triggers* [1], and with these three complementary features we are able to make considerable efficiency gains.

A key reason for the success of Boolean Satisfiability (SAT) solvers is their ability to propagate OR constraints efficiently, making use of movable triggers. We successfully generalise this approach to an OR of an arbitrary set of constraints, maintaining the crucial property that at most two constraints are active at any time, and no computation at all is done on the others. We also give an AND propagator within our framework, which may be embedded within the OR. Using this approach, we demonstrate speedups of over 10,000 times in some cases, compared to traditional constraint programming approaches. We also prove that the OR algorithm enforces generalised arc consistency (GAC) when all its child constraints have a GAC propagator, and no variables are shared between children. By extending the OR propagator, we present a propagator for ATLEASTK, which expresses that at least k of its child constraints are satisfied in any solution.

Some logical expressions (e.g. exclusive-or) cannot be compactly expressed using AND, OR and ATLEASTK. Therefore we investigate *reification* of constraints. We present a fast generic algorithm for reification using satisfying sets and movable triggers.

1 Introduction

Problems often consist of choices. Making an optimal choice which is compatible with all other choices made is difficult. *Constraint programming* (CP) is a branch of Artificial Intelligence, where computers help users to make these choices. Constraint programming is a multidisciplinary technology combining computer science, operations research and mathematics. Constraints are a powerful and natural means of knowledge

representation and inference in many areas of industry and academia, arising in in design and configuration; planning and scheduling; diagnosis and testing; and in many other contexts.

A *constraint satisfaction problem* (CSP [2]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. For example, the problem might be to fit components (values) to circuit boards (decision variables), subject to the constraint that no two components can be overlapping. An assignment maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* to a CSP is an assignment to all the variables which satisfies all the constraints. In this paper we consider solving CSPs through backtrack search with an inference step at each node [2].

Modelling is the process of representing a problem as a CSP. To allow natural modelling of some problems, the logical connectives of AND and OR are required between constraints. For example, in a school timetabling problem you may have either *Teacher1* OR (*Teacher2* AND *Teacher3*) taking a particular class. It is also sometimes useful to be able to apply NOT to a constraint, this is often done in CSP by means of *reification*. The reification of a constraint C produces another constraint C_r, such that C_r has an extra Boolean variable r in its variable set, and (in any solution) r is set to true if and only if the original constraint C is satisfied. In this paper we discuss the neglected area of how to efficiently implement these logical connectives across constraints, which are the fundamental building blocks of CSP models [3] (chapter 11).

During the search for a solution of a CSP, constraint *propagation* algorithms are used. These propagators make inferences, recorded as domain reductions, based on the domains of the variables constrained. If at any point these inferences result in any variable having an empty domain then search backtracks and a new branch is considered. Propagators and generalised arc consistency (GAC) are important concepts in this paper. When considering a single constraint C, GAC is the strongest possible consistency that a propagation algorithm can enforce. Enforcing GAC removes all domain values which are not compatible with any solution of C. In [3] (chapter 3), Bessiere defines GAC and discusses the complexity of enforcing it.

In this paper we consider propagating logical combinations of constraints. For example, for constraints C_1, C_2, C_3, C_4 we may wish to post the following expression and propagate it efficiently.

$$(C_1 \wedge C_2) \Rightarrow (C_3 \vee C_4)$$

It is desirable to make use of existing propagators for C_1, C_2, C_3 and C_4 since these may be highly efficient specialised propagators.

1.1 A Traditional Approach

A traditional approach (probably the most common) is to individually create *reified* propagators for the four constraints. These introduce an additional Boolean variable representing the truth of the constraint (e.g. the reified form of C_1 is the constraint $r_1 \Leftrightarrow C_1$, so in any solution r_1 is TRUE if and only if C_1 is satisfied). A logical expression can be enforced on the additional Boolean variables to obtain the desired combi-

nation. The example above translates into the following collection of constraints:[1]

$$r_1 \Leftrightarrow C_1, \quad r_2 \Leftrightarrow C_2, \quad r_3 \Leftrightarrow C_3, \quad r_4 \Leftrightarrow C_4, \quad (r_1 \wedge r_2) \Rightarrow (r_3 \vee r_4)$$

This scheme has three major disadvantages. First, it can be very inefficient because every reified constraint is propagated all the time. For example consider an OR of a set of n constraints. As we will demonstrate in Section 4, at most two constraints need to be actively checked at any time. However, a reification approach will propagate all n reified constraints at all times. Second, developing reified propagators individually for each constraint is a major effort. Third, when a variable occurs multiple times in an expression, the reified decomposition may propagate poorly. In this paper we address the first two issues but not the third: we achieve the same level of consistency as the reified decomposition.

1.2 Two Vital Features of a Solver for a New Approach

The key finding of this work is that two vital features of the solver must be combined to achieve efficient propagation of logical connectives. If either feature is not available, then the other is of limited benefit. The two features are *constraint trees*, which allow a *parent constraint* to control the propagation of its children, and *movable triggers* which allow a constraint to change the events [3] (ch. 14) it is interested in during search.

Consider an OR of n constraints over disjoint variable sets. We will show that at most two of the constraints need to be considered at any time, because if two of the constraints are satisfiable then no propagation can occur. Once two satisfiable constraints have been identified, all other constraints are presently irrelevant and no computation time should be wasted on them. This is essential to efficiency when n is large.

Constraint trees allow us to stop checking irrelevant constraints. However, this is not enough to achieve zero cost for irrelevant constraints: there is a cost to generate trigger events for the constraints. It is necessary to remove triggers not currently of interest, hence movable triggers are also required.

The following table summarises the costs caused by irrelevant constraints.

	Static Triggers	Movable Triggers
Reification	All reified constraints propagated at all times	All reified constraints propagated at all times
Constraint Trees	Trigger events received for all constraints at all times	Irrelevant constraints cause *no cost*

Our implementations are in the Minion solver [4], though the presentation is not specific to Minion.

1.3 Overview

There are a number of solver architecture decisions which impinge on propagating logical combinations of constraints. In Section 3 we describe three architecture features which are key to the new algorithms presented in this paper. *Satisfying sets* (Section

[1]In some solvers it would be necessary to further decompose $(r_1 \wedge r_2) \Rightarrow (r_3 \vee r_4)$.

3.3) are novel to the best of our knowledge. We also provide the first implementation of *constraint trees* (Section 3.2). Movable triggers [1] are also described in Section 3.1 to aid understanding of the rest of the paper.

In Section 4, we present a propagator for the constraint ATLEASTK, which ensures that at least k of a set of constraints are satisfied in any solution. Both AND and OR are special cases of ATLEASTK. Via the constraint trees framework, ATLEASTK constraints may be nested to any depth, and also may be reified using the algorithms given in Section 5. The ATLEASTK propagator maintains the crucial property that only $k+1$ constraints are checked (or k propagated) at any time — no computation at all is done on the others. Section 4 also contains experiments on the efficiency of Watched OR, AND and ATLEASTK, which demonstrate huge speedups in some cases.

In Section 5 we consider reification. Some logical expressions (e.g. exclusive-or) cannot be compactly expressed using only AND, OR and ATLEASTK, so a more general approach is needed. Therefore we investigate the use of satisfying sets, movable triggers and constraint trees for reification of constraints. To avoid implementing reified propagators for individual constraints, we developed four generic algorithms which can be used with any constraint C, provided that there is a propagator for $\neg C$ available. We compare algorithms which use satisfying sets and movable triggers with alternatives using static triggers, and again we demonstrate huge speedups in some cases.

Finally, the paper is concluded in Section 6.

2 Background

2.1 Preliminaries

A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined as a sequence of n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, a sequence of domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ where D_i is the finite set of all potential values of x_i, and a set $\mathcal{C} = \{C_1, C_2, \ldots, C_e\}$ of constraints. A literal is a pair $\langle x_i, d_i \rangle$, with $x_i \in \mathcal{X}$ and $d_i \in D_i$. An assignment A is a partial function $A : X \rightarrow D_{\text{all}}$ such that $A(x_i) \in D_i$, where $D_{\text{all}} = \bigcup_i D_i$. In a complete assignment A is a total function, i.e., every $x_i \in X$ is mapped by A.

Within CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a constraint $C_k \in \mathcal{C}$ consists of a sequence of $r > 0$ variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ (where $\mathcal{X}_k$ is the *scope* of the constraint) with respective domains $\mathcal{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle$ s.t. $\mathcal{X}_k$ is a subsequence[2] of $\mathcal{X}$ and $\mathcal{D}_k$ is the corresponding subsequence of $\mathcal{D}$. C_k has an associated set $C_k^S \subseteq D_{k_1} \times \ldots \times D_{k_r}$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_k$. A constraint is *satisfied* under a complete assignment to the variables iff the assigned values of $\mathcal{X}_k$ in sequence form a tuple in C_k^S. A *solution* to a CSP is a complete assignment which satisfies all the constraints.

The reified form ($r_k \Leftrightarrow C_k$) of a constraint C_k is satisfied iff r_k is assigned 1 and C_k is satisfied, or r_k is assigned 0 and C_k is not satisfied. The reifyimplied form ($r_k \Rightarrow C_k$) of C_k is satisfied iff r_k is assigned 0, or r_k is assigned 1 and C_k is satisfied.

The AND of a set of constraints is satisfied iff all constraints in the set are satisfied. The OR of a set of constraints is satisfied iff at least one of the constraints in the set is

[2]We use subsequence in the sense that $\langle 1, 3 \rangle$ is a subsequence of $\langle 1, 2, 3, 4 \rangle$.

satisfied. The ATLEASTK of a set of constraints (with parameter k) is satisfied iff at least k of the constraints in the set are satisfied.

A *subdomain* for a variable x is a subset of its initial domain. A subdomain list is a sequence of subdomains for distinct variables. A subdomain list $\langle D_1, \ldots, D_k \rangle$ *allows* the set of assignments corresponding to $D_1 \times \ldots \times D_k$.

A *propagator* for a constraint C is a function which takes a subdomain list for $\mathcal{X}_C$ and returns a new subdomain list, which does not allow any extra assignments, and does not remove any assignments which satisfy C. Further, if every subdomain in a subdomain list SD on $\mathcal{X}_C$ is singleton, then a propagator must empty all the domains iff C^s does not contain the single allowed assignment from SD.

A propagator is *GAC* if it removes every domain value possible without violating the definition of propagator. A complete discussion of propagators can be found in [3] (chapter 3).

Given a constraint c and a subdomain list SD for the variables in $\mathcal{X}_c$, then c is *disentailed* if every assignment allowed by SD does not satisfy c. c is *entailed* if every assignment allowed by SD satisfies c.

2.2 Related Work on Propagation Control

Brand and Yap [5] present a framework for reducing redundant propagation in logical combinations of constraints. This is named the *controlled propagation framework* (CPF). In CPF the standard reification approach is used, and improved by adding control flags to each constraint. For an individual reified constraint $C \Leftrightarrow b$, the value of b controls whether it is propagated (positively or negatively) and the control flags indicate whether it prunes b: the flags `chk-true` and `chk-false` indicate whether C is checked for entailment and disentailment. b is only pruned if at least one of `chk-true` or `chk-false` is present. The flag `irrelevant` indicates that $C \Leftrightarrow b$ need not be propagated at the current search node or its descendents. This would usually mean $C \Leftrightarrow b$ is revoked and its triggers removed. The control flags are manipulated using implication rules.

CPF is implemented in a Constraint Logic Programming context that allows constraints to be posted (and triggers added) as search moves forward. The posted constraints (and their triggers) are removed on backtracking. This allows parts of a decomposition to be generated as needed during search. Constraints may also be revoked (and restored on backtracking).

CPF has the same goals as our work. A detailed comparison is given in Section 3.4.

2.3 Related Work on OR

Many authors have considered *constructive disjunction* for propagating OR. For example, Müller and Würtz [6, 7] present a constructive disjunction algorithm implemented in Oz. Assuming that all child constraints have GAC propagators, constructive disjunction is able to enforce GAC over the OR, regardless of whether child constraints share variables. However, this is achieved by making a copy of the variable domains for each child constraint and propagating each child independently. A value which is pruned by every child constraint (i.e. pruned in each copy of the domains) is then pruned globally

by the OR. It is not clear that this algorithm can be implemented efficiently. Lagerkvist and Schulte [8] observed a performance penalty of over 45% when executing propagators on copies of the domains and mirroring the result back to the primary domains, compared with executing directly on the original variables.

Constructive disjunction may be valuable for problems where strong propagation of OR is required. However, in this paper we consider more lightweight methods that do not require duplication of variable domains. Therefore we consider constructive disjunction to be outside the scope of this paper.

Bacchus and Walsh [9] give some theoretical results about logical combinations of constraints, including AND, OR and negation. Concerning OR, the paper only states that the set of inconsistent values of the OR is the intersection of the inconsistent values of each child constraint. This would perform the same domain reductions as constructive disjunction. The authors give a basic algorithm but do not consider incremental propagation (which is vital for efficiency). Adapting this algorithm for incrementality would require tracking the state of variable domains independently for each child — essentially duplicating the variable domains. This would be equivalent to the algorithm of Müller and Würtz [6, 7].

Lhomme [10, 11] presents an alternative to constructive disjunction which performs the same domain reductions. Lhomme's algorithm is claimed to be more efficient than constructive disjunction. It is based on finding satisfying assignments (represented as tuples of values) for the constraints. Each relevant variable-value pair is *supported* by a satisfying tuple for one of the constraints in the disjunction, or it is pruned.

While Lhomme's algorithm may be faster than constructive disjunction, it maintains a large set of supporting tuples (one for each variable-value pair where the variable is shared between two child constraints). Our proposed algorithm maintains only two partial tuples (and enforces a weaker consistency), therefore it is much more lightweight.

In SAT, the constraints (OR of Boolean literals) are often propagated by *2-literal watching* [12]. This scheme has the advantage that only two literals are active at any time, and the others incur no cost. One of the major contributions of this paper is our proposed algorithm Watched OR in Section 4, which shows how the basic techniques behind 2-literal watching can be efficiently extended to support arbitrary constraints and propagators, keeping the efficiency which comes from only having two active disjuncts at any time.

2.4 Related Work on Reification

Reification of a constraint C produces the constraint $r \Leftrightarrow C$, where r is a Boolean variable. We focus on generic approaches to reification that can be applied to any constraint that has the appropriate algorithms defined for it. For example, we prove that a generic reification algorithm that enforces GAC efficiently requires GAC propagators for both the constraint and its negation.

Indexicals (proposed by Van Hentenryck et al. [13]) allow simple propagators to be specified in a high-level language. They can be extended slightly to allow reification [3] (Section 14.2.6). However, it is not possible to express polynomial-time GAC propagators for constraints such as AllDifferent [14] in the indexicals language.

Propia [15] allows constraints to be expressed as Prolog predicates. The predicate specifies the constraint semantically as opposed to giving a propagator for the constraint. To implement reification, a predicate would be required for both the constraint and its negation. Similarly to indexicals, it is not possible to specify sophisticated propagators in propia, therefore it does not offer an efficient generic solution.

Schulte proposed a generic reification algorithm [16] based on the concept of *computation spaces*. A computation space is an isolated environment which allows a propagator to be executed without affecting the primary variables. The computation space includes independent variable domains. For $r_i \Leftrightarrow C_i$, C_i is posted in the space, and propagated. If it fails, then $r_i \neq 1$ (i.e. 1 is pruned from r_i). If it is entailed (i.e. equivalent to the constraint TRUE), then $r_i \neq 0$. If $r_i = 1$ then the effects of propagating C_i are copied to the primary variables. In the case where $r_i = 0$, there is no propagation of $\neg C_i$, and the algorithm does nothing until C_i is entailed. The approach later proposed by Lagerkvist and Schulte [8] is virtually the same algorithm implemented with propagator groups.

Both these approaches have the disadvantages that they duplicate variables and do not propagate the negative constraint $\neg C$ when $r = 0$. Lagerkvist and Schulte compared a hand-implemented reified constraint to the generic algorithm. The generic algorithm was substantially slower, with the solver taking between 29% and 106% extra time [8].

The commercial product ILOG Solver implements reification, but we found no literature describing the algorithm.

In Section 5 we propose new reification algorithms which avoid the overhead of duplicating variables, while also being able to encapsulate any propagator, unlike indexicals or propia.

2.4.1 Triggering

Given a propagator P for a constraint c, there may be many subdomain lists SD where $P(SD) = SD$. In these situations, it is not necessary to run the propagator. Rather than invoke propagators on any domain change, solvers provide a list of *events*, which propagators can subscribe to. A propagator subscribes to an event by attaching a *trigger* to it. *Executing* a trigger calls the propagator which generated it, with a reference to the event which occurred (we refer to this as *triggering* the propagator). When events occur they are placed in a queue. Items in the queue are processed by executing every trigger on that event in turn.

The exact set of events which can be subscribed to varies between solvers. For any variable x with domain $D(x)$ and domain value i, Minion supports the following events: i is removed from $D(x)$; any value is removed from $D(x)$; x is assigned; the maximum value of $D(x)$ removed; and the minimum value of $D(x)$ removed.

The propagator must always be triggered when $P(SD) \neq SD$; otherwise the propagator would fail to enforce the correct level of consistency.

2.4.2 Constraint Trees

There are highly efficient propagators already written for many constraints. We would like to be able to combine these propagators to build new propagators. Constraint trees provide a highly efficient framework for achieving this goal.

To define constraint trees, we define the concept of *meta-variables*. These are not CSP variables, and have none of the associated overhead, but are merely for pedagogical purposes. Meta-variables are Boolean, and therefore have three states (0, 1, and unset). The current subdomain of a meta-variable x^m for a constraint c denotes if, in the current subdomain list, c is entailed $(x^m = 1)$, disentailed $(x^m = 0)$ or neither $(x^m \in \{0, 1\})$. The state of x^m is a property of c, it is never stored.

A constraint tree is a rooted tree $T = \langle V, E, r \rangle$ with root $r \in V$. Each node $b \in V$ has associated with it a constraint and a meta-variable. The scope of the constraint on node b may contain both CSP variables and the meta-variables of the children of b. A constraint tree as a whole is satisfied iff the constraint associated with r is satisfied. A constraint tree is a type of constraint, and may be contained in a CSP (defined in Section 2.1). Constraint trees are not a novel concept (for example, Bacchus and Walsh make use of them [9]), however they allow us to define a novel propagation framework.

Now we discuss propagation of constraint trees. The constraint for every vertex $a \in V$ has a propagator P_a associated with it. Propagators at leaves of the tree are conventional propagators as described in Section 2.1. The propagator P_b for an internal vertex $b \in V$ of the tree is able to prune its CSP variables, query any child constraint for disentailment, and invoke the propagator for any child constraint.

By an abuse of notation, we refer to the vertices in T as constraints. The *parent* of the constraint attached to node a is the parent of a in T. The children of the constraint attached to a node $a \in V$ are the children of a in T.

Constraint trees are used to implement constraints that are expressed as logical combinations of other constraints. For example, $(x = 1) \vee ((y = 2) \wedge (x = 3))$ could be represented by the constraint tree in Figure 1. We will present propagators for the interior nodes $\wedge$ and $\vee$ in Section 4.

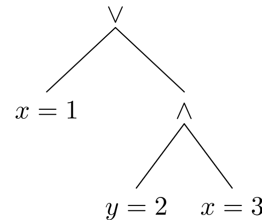


Figure 1: Constraint tree for $(x = 1) \vee ((y = 2) \wedge (x = 3))$

One issue which is often ignored when discussing propagation algorithms is when variables are repeated within the constraint. Most propagation algorithms that enforce GAC will not enforce GAC when variables are repeated. Many constraints with polynomial-time GAC propagators become NP-hard to enforce GAC once repeated variables are taken into account, for example the Global Cardinality Constraint [17]. We address the issue of repeated variables separately for each of the proposed algorithms in this paper.

3 Solver Architecture

In order to implement logical connectives efficiently, we made a number of solver architecture decisions which are described in this section.

3.1 Movable Triggers

One important part of how propagators are implemented is how they are triggered, as described in Section 2.4.1. In Minion there are three classes of triggers, discussed in depth in [1]. In this paper we exploit both static and movable triggers. In [1], movable triggers are referred to as *watched literals*.

Static: These triggers are placed on variables at the beginning of search. They can never be moved or removed.

Movable: These triggers can be placed, moved and removed during search. When search backtracks, they are **not** restored to their previous place.

Using movable triggers can produce great improvements in the performance of the solver, as observed in SAT [12] and CSP [1]. Some solvers support movable triggers which backtrack during search. The algorithms described in this paper can be trivially modified to work with such solvers, at the cost of the extra overhead of backtracking the triggers and data structures.

3.2 Constraint Trees

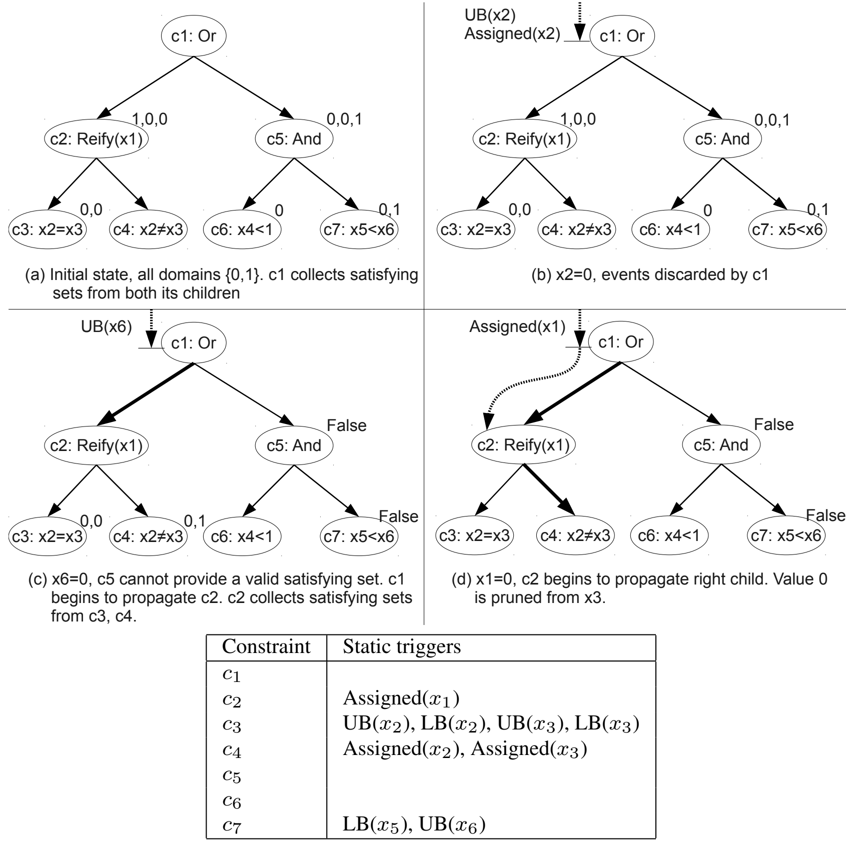
All the following algorithms use the concept of a Constraint Tree, as defined in Section 2.4.2. In this paper we assume that a parent constraint controls when a child is propagated, and also every constraint has a method which can detect if a constraint is *disentailed*. We will show in Section 4 how we can detect disentailment of interior nodes by using the disentailment detectors of their children.

Any constraint which has a propagator and disentailment checker can function as a child constraint, allowing us to leverage the large number of already implemented highly efficient propagators in the CP literature. All the parent constraints we describe in this paper can also function as a child of another constraint.

Static triggers are handled as follows in a tree of constraints. At setup time, all propagators in the tree place the static triggers that they need. During search, all trigger events are passed to the topmost propagator. Each parent propagator passes the appropriate trigger events through to the children which are currently propagating, and discards others. An example of this is shown in Figure 2. Three assignments occur in sequence ($x_2 = 0$, $x_6 = 0$ and $x_1 = 0$) and the corresponding events are passed to the propagator of c_1 by the solver core. In Fig 2(b), c_1 is propagating neither of its children so it discards the two events. In (c), c_1 is propagating its left child, but the trigger event belongs to the right child so it is discarded. In (d), c_1 passes the trigger event on to c_2 because c_2 is currently propagating.

Movable triggers are somewhat more complicated, but they allow triggers for non-propagating children to be removed, reducing the number of unnecessary trigger events. Operations on movable triggers are described in detail with the algorithms in sections 4 and 5.

For both classes of trigger, the trigger events are passed in at the top of the tree, and filter down, which adds a small overhead to propagating the constraints at the leaves of the tree. However, once the propagators are invoked they execute as if they were not within a constraint tree, directly reading and changing the subdomain of variables.



Constraint	Static triggers
c_1	
c_2	Assigned(x_1)
c_3	UB(x_2), LB(x_2), UB(x_3), LB(x_3)
c_4	Assigned(x_2), Assigned(x_3)
c_5	
c_6	
c_7	LB(x_5), UB(x_6)

Figure 2: Example of static trigger events and satisfying sets in a constraint tree representing $(x_1 \leftrightarrow x_2 = x_3) \vee (x_4 < 1 \wedge x_5 < x_6)$. Satisfying sets are written beside constraints as a list of values. For example, the satisfying set for c_5 is $\langle x_4, 0\rangle, \langle x_5, 0\rangle, \langle x_6, 1\rangle$. Movable trigger events are omitted from this diagram. In the table at the bottom, the static triggers placed by each constraint are listed.

3.3 Satisfying Sets

In many of the algorithms in this paper, we want a fast method of checking if a constraint is satisfiable. One way of doing this is to execute its propagator and check to see if it removes all the values from the domain of any variable. However this is clearly inefficient because it computes domain deletions as well as deducing whether the constraint is satisfiable. In this section, we introduce satisfying sets, a simple and efficient framework for checking disentailment.

Definition 1. *Given a constraint C, a satisfying set is a set of literals F from $\mathcal{X}_C$ such that every assignment to $\mathcal{X}_C$ which contains all the literals in F also satisfies C. A satisfying set F is* complete *if, additionally, every subdomain list for $\mathcal{X}_C$ that contains all the literals in F allows* at least one *assignment that satisfies C.*

Example 2. *Consider the constraint $X + Y + Z \geq 2$, for variables X, Y and Z with domains $\{0, 1\}$. The set of literals $\{\langle X, 1\rangle, \langle Y, 1\rangle\}$ is a complete satisfying set. It is a satisfying set as the two assignments that contain these literals, $\langle X, Y, Z\rangle = \langle 1, 1, 0\rangle$ and $\langle 1, 1, 1\rangle$, both satisfy the constraint. It is complete because any subdomain list for $\mathcal{X}_C$ which contain this satisfying set must contain an assignment where $X = 1$ and $Y = 1$. Therefore regardless of the assignment to Z the sum of the variables must be greater than or equal to two.*

The set of literals $\{\langle X, 0\rangle, \langle X, 1\rangle\}$ is trivially a satisfying set, as there can be no assignment which contains both of these literals, as they are from the same variable. It is not complete, because the subdomain list $X \in \{0, 1\}$, $Y, Z \in \{0\}$ does not contain an assignment which satisfies the constraint.

Given a satisfying set for a constraint, we know that if none of the literals in the satisfying set are removed, we cannot end up in a state where every variable is assigned and the constraint is not satisfied. This basic guarantee will be used to ensure algorithms using satisfying sets are correct. A complete satisfying set produces a much stronger guarantee, that the constraint is never disentailed as long as no literal from the satisfying set is removed. This will be necessary for any propagator which makes use of satisfying sets to enforce GAC.

Definition 3 introduces the concept of a satisfying set generator.

Definition 3. *A satisfying set generator for a constraint C is a function that takes a subdomain list SD and either returns a satisfying set within SD, or* FAIL. *A satisfying set generator may only return* FAIL *when there is no assignment within SD that satisfies C.*

A satisfying set generator is complete *if it only returns complete satisfying sets. This implies it must return* FAIL *exactly when there is no assignment within SD that satisfies C.*

One question is for which constraints satisfying set generators can be implemented in polynomial time, and when they can be made complete. Definition 4 presents the trivial satisfying set generator, which provides a polynomial-time satisfying set generator for any constraint. Lemma 5 shows that the trivial satisfying set generator is valid. Notice that an incomplete satisfying set must contain two literals of the same variable, by Definition 1. The following trivial satisfying set generator makes use of this fact.

Definition 4. *The trivial satisfying set generator for a constraint C and subdomain list SD is defined as follows:*

1. *There exists an $X \in \mathcal{X}_C$ such that $|SD(X)| > 1$: Return a satisfying set containing two literals from $SD(X)$.*
2. *SD allows exactly one assignment: If this assignment satisfies C, return the satisfying set containing all the literals in SD, else return* FAIL.

Lemma 5. *For every constraint C, the trivial satisfying set generator is valid and runs in polynomial time.*

Proof. Any set of literals that contains two assignments to one variable is a satisfying set, as no assignment can contain two values for one variable. Once all variables are assigned, the trivial satisfying set generator either returns FAIL, or returns a complete assignment which must satisfy C. The complexity result is trivial. □

Theorem 6 categorizes the complexity of complete satisfying set generators.

Theorem 6. *A constraint C has a polynomial time complete satisfying set generator if and only if it has a polynomial time GAC propagator.*

Proof. Given a complete satisfying set generator, it is possible to check if a subdomain list for $\mathcal{X}_C$ contains a satisfying assignment, by seeing if the satisfying set generator returns FAIL. Lemma 1 of [17] proves this is polynomially equivalent to having a GAC propagator. Alternatively, given a GAC propagator for C, we can construct a complete satisfying set generator as follows.

The GAC propagator will empty the domains of the variables if the subdomain list contains no assignment which satisfies C, which is exactly the situation in which a complete satisfying set generator must return FAIL. Assuming the GAC propagator does not empty the domains, then they must contain at least one satisfying assignment. A complete assignment which satisfies C is a complete satisfying set, as any subdomain list which contains it contains a satisfying assignment. The following algorithm produces an assignment which satisfies C within $|\mathcal{X}_C|$ invocations of the propagator: (1) Run the GAC propagator. (2) If any unassigned variable exists, choose one and assign it to any value in its current subdomain. (3) If any variable is unassigned, return to step 1. □

While Theorem 6 shows how to build a complete satisfying set generator from a GAC propagator, for many constraints there is a faster complete satisfying set generator which produces smaller satisfying sets. Theorem 6 always returns a complete satisfying set with as many literals as variables in $\mathcal{X}_C$. The complexity of finding smaller complete satisfying sets is an open problem which we leave for future work. For all the constraints in Minion which have GAC propagators, we have constructed complete satisfying set generators (often by taking a small part of the propagator). We present two cases here as examples.

Example 7. *Consider the constraint $M[x] = y$ for an array of variables M and variables x and y. Given a subdomain list, this constraint is satisfiable if and only if there exist i and j such that i is in the subdomain of x and j is in the subdomain of both $M[i]$ and y. If such i and j exist, then the literals $\langle x, i\rangle$, $\langle M[i], j\rangle$ and $\langle y, j\rangle$ form a complete satisfying set.*

Example 8. *The complete satisfying set generator given in Theorem 6 requires finding a complete assignment which satisfies the constraint. The first part of the AllDifferent propagator [14] finds a satisfying assignment, so a complete satisfying set generator can be formed by truncating the algorithm at this point.*

3.4 Comparison to Previous Work

The most closely related previous work is by Brand and Yap [5] (*CPF*, described in Section 2.2). There are several important differences between our work and CPF. Firstly, their approach has reification variables whereas with constraint trees we are able to avoid them (to reduce overheads). Secondly, constraint trees are static whereas their framework posts new constraints and revokes constraints during search, potentially saving space but with a time overhead. Thirdly, our approach is restricted to propagation and checking of child constraints. It is not possible to propagate the negation of a child, except by introducing the negation as another child constraint. CPF allows propagation and checking of the negation of any constraint, at the cost of requiring a propagator for the reified form of each constraint.

Fourthly, and perhaps most importantly, CPF does not make use of movable triggers. Constraints are posted and revoked during search, but between these two events the triggers are fixed. The framework proposed here makes extensive use of movable triggers (with satisfying sets) to check disentailment of both primitive and parent constraints. For example, checking disentailment of the AllDifferent constraint requires a movable trigger on one literal per variable. By contrast, in CPF it would be necessary to check the constraint for every domain change[3].

It is possible to implement 2-literal watching for a SAT clause on Boolean variables in CPF. This is done by dynamically posting literals such that only two are checked for disentailment at any time. The same technique can be applied to a disjunction of constraints, however CPF is not able to combine the technique with movable triggers and satisfying sets for efficient disentailment checking of the child constraints.

4 Efficient Propagators for AtLeastK, And and Or

In this section we present a new algorithm for AtLeastK of a set of constraints (defined in Section 2.1). Then we show how that algorithm can be specialized to And and Or.

4.1 Theoretical Overview

The AtLeastK algorithm is defined as: AtLeastK(k, Con) is true if at least k constraints in Con are true. In Theorem 9, we show the fundamental result which the algorithm for AtLeastK uses to enforce GAC, assuming all the constraints in Con have a GAC propagator and no two constraints in Con share a variable.

Theorem 9. *Let constraint $C =$ AtLeastK$(k, \{Con_1, Con_2, \ldots, Con_n\})$ for a constant k and constraints Con_i, where the scopes of the Con_i are disjoint.*

Given a subdomain list SD for $\mathcal{X}_C$, where the subdomain of every variable in SD is non-empty, then SD is GAC with respect to C if and only if, either:

1. *At least $k+1$ of the Con_i have satisfying assignments in SD; or*

[3]Assuming that complete satisfiability checking is required.

2. *exactly k of the Con_i has a satisfying assignment in SD, and SD is GAC with respect to each of these k constraints.*

Proof. 1. Assume that there exists some set S such that $|S| = k + 1$ and Con_i has a satisfying assignment for every $i \in S$. Then given any assignment to any variable there are at least k members of S which do not contain this variable in their scope. A satisfying assignment to C can be generated by assigning these k constraints a satisfying assignment, and then assigning all other variables any value. Therefore, every assignment to every variable is supported.

2. Assume there exists a set S such that $|S| = k$ and Con_i is satisfiable if and only if $i \in S$. This implies in any satisfying assignment to C, then every Con_i for $i \in S$ must be satisfied. Therefore for each $i \in S$, any assignment to any variable in the scope of Con_i which cannot be extended to a satisfying assignment to Con_i must be removed. This is the definition of $GAC(Con_i)$.

   Any variable not in the scope of any Con_i for $i \in S$ can be assigned any value.

If there are less than k members of the Con which are satisfiable, clearly no assignment can satisfy C. □

4.2 The Watched ATLEASTK Propagator

Our algorithm is split into three distinct phases, namely a *setup* phase, a *watching* phase and a *propagation* phase. In this section we will present each phase separately. Before presenting the steps in our algorithm, we first describe the state that the algorithm stores between calls.

PropagateMode: a Boolean which represents if we are in the propagation phase of the algorithm. It is reverted when search backtracks.

Watches: The indices of the $k + 1$ child constraints that are currently being watched. These are not reverted when search backtracks.

The algorithm operates on child constraints c_1 to c_n, which are each required to have a propagator and a satisfying set generator. By using the constraint trees framework (Section 3.2), the child propagators are able to use any kind of trigger available in Minion, and executing them is almost as efficient as propagating an ordinary constraint (the only overhead being passing trigger events through the ATLEASTK).

The algorithm begins in the setup phase. This searches for $k+1$ satisfiable children. If $k+1$ can be found then they are all watched. If exactly k are found then the propagation phase is entered, and if fewer than k are found then the constraint fails, signalling that search should backtrack. The code for the setup phase is shown in Algorithm 1.

PropagateMode = FALSE;
if $\exists S \subseteq \{1 \dots n\}.\ (|S| = k+1\ \wedge\ \forall c_i \in S.\ c_i$ *has satisfying set*$)$ **then**
  Place movable triggers on satisfying sets of all $c_i \in S$;
  Watches=S
else
  if $\exists S \subseteq \{1 \dots n\}.\ (|S| = k\ \wedge\ \forall c_i \in S.\ c_i$ *has satisfying set*$)$ **then**
    Initialise propagation of c_i for all $i \in S$;
    PropagateMode = TRUE;
  else
    Fail;
  end
end

Algorithm 1: Code for setup phase

While PropagateMode is FALSE, whenever a literal of a satisfying set is pruned, the watching phase of the algorithm is called. This either finds a new satisfying set, or (if only k children are satisfiable) starts to propagate all k satisfiable children. The code for the watching phase is shown in Algorithm 2

Input: i : The satisfying set of constraint c_i has been lost
Global Data: PropagateMode
if *PropagateMode* **then**
  Return
end
if c_i *is satisfiable* **then**
  Move movable triggers to new satisfying set of c_i;
else
  if $\exists k.\ c_k$ *is satisfiable and* $k \notin$*Watches* **then**
    Move movable triggers to satisfying set of c_k from c_i;
    Watches=$(\text{Watches} \setminus \{i\}) \cup \{k\}$;
  else
    $\text{Prop} = \text{Watches} \setminus \{i\}$;
    Initialise propagation of c_j for all $j \in \text{Prop}$;
    PropagateMode = TRUE;
  end
end

Algorithm 2: Code for watching phase

Finally, the propagation phase is active when PropagateMode is TRUE. All trigger events belonging to any c_j where $j \in \text{Prop}$ are passed through to c_j. We do not give code for the propagation phase.

It is possible to receive stale trigger events from movable triggers which were placed in a different phase because movable triggers are not backtracked. Therefore in the watching and propagation phases, some trigger events must be ignored or otherwise handled specially. These are listed below, but in Algorithms 1 and 2 we assume such events have already been dealt with appropriately.

Watching Phase: Trigger events from the propagation phase may be received in this

phase. Movable triggers are removed and ignored; static triggers cannot be removed and are just ignored.

Propagation Phase: Static trigger events for children not being propagated are ignored. All trigger events from setup and watching phases are ignored, and movable trigger events from a non-propagating child cause the corresponding movable trigger to be removed.

To prove our algorithm correct, we present two invariants. These two invariants are exactly the conditions which are required to enforce GAC on ATLEASTK from Theorem 9, so our algorithm achieves GAC under the assumptions that all children have GAC propagators, all satisfying set generators are complete and no pair of children share variables.

Lemma 10. *After the setup phase for the algorithm has completed, at any point during search where failure has not occurred and all items on the constraint queue have been executed, the following two invariants are true.*

1. *PropagateMode* = FALSE *implies that* $k + 1$ *satisfying sets of* $k + 1$ *child constraints are being watched.*
2. *PropagateMode* = TRUE *implies that* $n - k$ *child constraints are known to be unsatisfiable, and the other* k *are being propagated.*

Proof. Invariant 1 is true after setup, and whenever search progresses forward. However, we must consider what happens when search backtracks. If PropagateMode was TRUE and remains so, then the condition is trivially true. There are two other cases to consider.

- Backtrack from node A where PropagateMode is FALSE to node B, where PropagateMode is still FALSE. The $k+1$ satisfying sets from A are retained, and they are valid at B since the domain sets at B are (non-strict) supersets of those at A.
- Backtrack from node A where PropagateMode is TRUE to node B where it is FALSE. The $k + 1$ satisfying sets were found at node B or at an intermediate state between A and B. They remain valid at B since the domain sets at B are supersets of those at any intermediate state.

For invariant 2, in both places where PropagateMode is set to TRUE, the invariant holds. Suppose PropagateMode is set to TRUE at node A. For all nodes B below A in the search tree, domain sets are a subset of those at A and therefore the invariant still holds (i.e. the $n - k$ unsatisfiable children remain unsatisfiable at B). When backtracking from A, PropagateMode is reverted to FALSE therefore the invariant holds. □

As stated above, the proof assumes that each child constraint has a GAC propagator and satisfying set generators are all complete. If this is not the case, in invariant 2 when k children are propagated the guarantee of GAC is lost, however it is clear that the algorithm remains correct as long as the children have propagators meeting the definition in Section 2.1.

The proof also assumes that no variables are shared between children. However, this assumption will often not be met, so it requires some discussion. In terms of constraint trees, consider an ATLEASTK vertex with two child vertices a and b, with associated meta-variables x^a and x^b, and associated constraints $y = 1$ and $y = 3$ on variable $y \in \{1, 2, 3\}$. Both meta-variables will be unassigned, and the ATLEASTK algorithm will not deduce that they cannot both be equal to 1 at the same time. The level of consistency enforced on the ATLEASTK is the same as achieved by doing the following: replace $y = 3$ with $y' = 3$ for a new variable y' of identical domain, add the new (GAC) constraint $y = y'$, and apply the Watched ATLEASTK algorithm. In this translation, the two children $y = 1$ and $y' = 3$ no longer share variables so Watched ATLEASTK enforces GAC.

In general the level of consistency enforced on an ATLEASTK with shared variables is the same as achieved by doing the following: reformulating the ATLEASTK to remove shared variables by duplicating variables and adding GAC equality constraints, then enforcing GAC on the reformulated ATLEASTK. It is assumed here that each child constraint has a GAC propagator, and that if a child constraint itself has repeated variables in its scope, that its propagator still enforces GAC.

As discussed in Section 2.3, previous work has shown that given a disjunction of constraints, each of which has a polynomial-time GAC propagator, it is possible to achieve GAC propagation over the whole disjunction in polynomial-time, even if disjuncts share variables (*constructive disjunction*). We leave efficiently combining constructive disjunction, constraint trees and satisfying sets to future work.

4.2.1 A Satisfying Set Generator for ATLEASTK

Our Watched ATLEASTK algorithm uses satisfying sets extensively. To be able to use ATLEASTK as a non-root node in a constraint tree, it is necessary to also have a satisfying set generator for ATLEASTK.

Definition 11. *Given satisfying set generators for a set* $\{c_1, \dots, c_n\}$ *of constraints, the satisfying set generator for* ATLEASTK $(c_1, \dots, c_n)$ *is defined as follows:*

If the satisfying set generators of more than $n - k$ *children return* FAIL*, then return* FAIL. *Otherwise choose any set of* k *children whose satisfying set generators do not return* FAIL*, and return the union of the satisfying sets they generate.*

Lemma 12. *The satisfying set generator for* $C =$ ATLEASTK$(c_1, \dots, c_n)$ *given in Definition 11 is correct. Further, it is complete if the satisfying set generators for the* c_i *are complete and for all* $i \neq j$, $\mathcal{X}_{c_i}$ *and* $\mathcal{X}_{c_j}$ *are disjoint.*

Proof. (Correct) The satisfying set generator for C returns FAIL when fewer than k children are satisfiable, matching the definition of ATLEASTK. A satisfying set F generated for C must contain satisfying sets for at least k of its children, therefore any assignment that contains F must satisfy those k children.

(Completeness) If C is unsatisfiable, there cannot exist k children of C which have a complete satisfying set, and so the satisfying set generator for C will return FAIL. If C is satisfiable, it must have at least k satisfiable children, so complete satisfying sets can be generated for these k children. Given any subdomain list SD for $\mathcal{X}_C$ which

contains these k complete satisfying sets, the same k children must have a satisfying assignment in SD. Joining these k disjoint assignments, together with any assignment to every other variable in $\mathcal{X}_C$, produces a satisfying assignment to C. Therefore the satisfying set generator for C is complete as long as the satisfying set generators for its children are complete, and the scopes of the children are disjoint. □

4.3 The Watched OR and AND Propagators

It is easy to take our algorithm for ATLEASTK, and generate algorithms for both AND and OR. OR is logically identical to ATLEASTK when $k = 1$, although of course this algorithm will not achieve constructive disjunction, as discussed in Section 2.3. Furthermore AND is equivalent to ATLEASTK when k is set equal to the number of children. (Watched AND is useless in isolation, as it will achieve identical propagation as posting the child constraints individually. However, it is useful as a child of an OR or ATLEASTK constraint.)

Given this observation, the propagators and satisfying set generators for Watched OR and Watched AND are straightforward specializations of Watched ATLEASTK. For both algorithms, there are some simplifications and performance gains which can be achieved by fixing k.

The algorithm for OR is a generalization of unit propagation (with 2-literal watching) in SAT [12]. A SAT clause is an OR of literals of Boolean variables ($\langle x_i, 0\rangle$ or $\langle x_i, 1\rangle$).

4.4 Complexity

Lemma 10 showed that the polynomial-time algorithm given for ATLEASTK achieves GAC if there are no repeated variables, so the specialisations of it we describe in Section 4.3 for Watched OR and Watched AND will also run in polynomial time and achieve GAC. As discussed previously in Section 2.3, it is possible to achieve GAC in polynomial time on Watched OR, even with variables repeated in different disjuncts. Lemma 14 shows that this is not the case for ATLEASTK for $k \geq 2$.

Definition 13. *The* TRUE *constraint is the constraint on no variables which contains a single empty tuple. Therefore it is always satisfied. The* FALSE *constraint is the constraint on no variables which contains no tuples, and is therefore always false.*

Lemma 14. *GAC propagation of* AND (C) *is NP hard if* $|C| \geq 2$*, and* ATLEASTK (k, C) *is NP-hard if* $k \geq 2$ *and* $|C| \geq 2$*.*

Proof. Bacchus et al. [9] show that the AND of two constraints with polynomial-time GAC propagators is NP-hard, when the constraints are allowed to share variables. We can extend this result to AND (C) for $|C| \geq 3$ by adding $|C| - 2$ copies of the TRUE constraint (Definition 13). We can further extend this result to ATLEASTK (k, C) by adding $k - 2$ copies of the TRUE and $|C| - k$ copies of FALSE (Definition 13). □

4.5 Experimental Results

We claimed in Section 1.2 that both constraint trees and movable triggers are essential for propagation of logical connectives. Here we test that claim on four different problems.

All of our experiments use Minion version 0.10 which can be downloaded from `http://minion.sourceforge.net`. We ran our experiments on 4 servers using Intel Xeon 2.4GHz CPUs. Each server has 2 cores and 2GB of memory, and is running Linux kernel 2.6.18. We repeated each experiment 5 times and took the instance with minimum runtime as representative, since this is the run suffering from the least interference and hence most closely approximating the ideal. The maximum coefficient of variation of any set of 5 runs is under 1.3%. We have published all our problem instances as online supplementary data associated with this paper at INSERT LINK[4]. All times are given in seconds.

4.5.1 The Generalised Pigeon-Hole Problem

The first experiment is a generalisation of the pigeon-hole problem. We consider the problem of finding assignments to a two-dimensional array of variables, where every pair of rows in the array must be unequal.

The parameters for this problem are the number of rows n, the number of columns p, and the domain size d. We introduce a matrix of variables $M[1 \ldots n, 1 \ldots p] \in \{1 \ldots d\}$. All five models must introduce $n(n-1)/2$ not-equal constraints between pairs of rows in M. We compare five representations of the constraint that two rows r_1 and r_2 are not equal.

Watched OR: Implement $M[r_1, 1] \neq M[r_2, 1] \vee \ldots \vee M[r_1, p] \neq M[r_2, p]$ as a Watched OR (described in Section 4.3) that enforces GAC.

Element: We ensure that r_1 and r_2 differ at some position by adding, for each pair of rows:

- New variables $X \in \{1, \ldots, p\}$ and $Y, Z \in \{1, \ldots, d\}$.
- The constraints $M[r_1, X] = Y$, $M[r_2, X] = Z$ and $Y \neq Z$

Sum: Decompose the model for **Watched OR** into:

- New variables $N[1 \ldots p] \in \{0, 1\}$.
- The constraints $\forall i.\ (N[i] \iff (M[r_1, i] \neq M[r_2, i]))$ and $(\sum N) \geq 1$

Watched Sum: The same model as **Sum**, except the constraint $(\sum N) \geq 1$ is replaced by a SAT clause implemented using movable triggers.

Custom: A custom-written propagation algorithm using static triggers on all variables, enforcing the same level of consistency as **Watched OR** (GAC).

[4] This link to be supplied by Elsevier at a later date

$\langle n, p, d \rangle$	**Element**		**Watched OR**	
	Time	Nodes	Time	Nodes
$\langle 8, 3, 2 \rangle$	25.81	12,335,593	0.27	25
$\langle 8, 3, 3 \rangle$	8,028.62	3,112,501,760	0.27	28
$\langle 8, 4, 2 \rangle$	2,137.13	1,092,789,218	0.27	33
$\langle 8, 4, 3 \rangle$	$>$109,067.70	$>$45,000,000,000	0.27	36

Table 1: Search size for small instances of the array pigeonhole problem

These models explore all four possibilities of using static or movable triggers, with reification or constraint trees, as shown in the table below.

	Static Triggers	Movable Triggers
Reification	**Sum**	**Watched Sum**
Constraint Trees	**Custom**	**Watched OR**

Note that using Theorem 6.6 from [18], as long as we get GAC on each of the constraints in the **Sum** and **Watched Sum** models, we get GAC over the whole OR, and further as long as we place the new variables at the end of the search ordering, the resulting searches will be identical to the **Watched OR** model. Therefore, the only model which could result in a different sized search is **Element**.

Since we achieve GAC, there is no scope for Lhomme's algorithm [10, 11] (or other constructive disjunction algorithms) to enforce a stronger consistency. Lhomme's algorithm is statically triggered, and would be similar to **Custom** in this context.

Table 1 shows just how badly the **Element** model performs in practice on some very small instances, quickly leading to insolvable problems which the other models we consider are all able to solve in less than a second. Due to the very poor performance of this model, it will not be considered further.

As the remaining four models produce identical search trees, in Table 2 we compare them on various instances in terms of the number of nodes of search they perform per second. The **Custom** model improves significantly on **Sum** and **Watched Sum** by eliminating the additional variables, but **Watched OR** is always faster than **Custom**, sometimes by several orders of magnitude. When in the watching phase, the Watched OR algorithm will use only four movable triggers: two for each watched child constraint. By comparison, the custom algorithm has assignment triggers on all variables. This illustrates the importance of using an appropriate triggering mechanism, in this case movable triggers.

With domain size 2, the Watched OR algorithm sometimes increases in speed as instance size increases. This surprising result is caused by a decrease in the proportion of variables with a movable trigger on them.

The small differences between **Sum** and **Watched Sum** show that the gain from using movable triggers for the sum constraint is often insignificant compared to the cost of propagating the reified not-equal constraints.

In summary, these results support the hypothesis that both constraint trees and movable triggers can be used to efficiently propagate OR.

$\langle n, p, d\rangle$	Watched OR	Sum	Watched Sum	Custom
$\langle 100, 5, 2\rangle$	191,536.22	19,304.05	29,404.22	54,180.04
$\langle 100, 10, 2\rangle$	499,007.21	1,268.15	1,377.21	79,704.14
$\langle 100, 20, 2\rangle$	1,576,413.85	755.48	782.40	87,443.99
$\langle 100, 30, 2\rangle$	1,579,347.99	548.23	564.70	84,170.60
$\langle 100, 40, 2\rangle$	1,461,316.06	424.32	428.23	78,234.20
$\langle 100, 50, 2\rangle$	1,439,796.97	370.62	373.95	76,766.77
$\langle 100, 5, 10\rangle$	690,482.51	1,404.05	1,439.98	105,234.71
$\langle 100, 10, 10\rangle$	379,255.81	817.18	838.69	103,457.50
$\langle 100, 20, 10\rangle$	239,937.97	378.53	385.92	79,418.76
$\langle 100, 30, 10\rangle$	203,991.09	266.58	307.82	71,914.23
$\langle 100, 40, 10\rangle$	155,887.24	234.83	253.54	65,572.34
$\langle 100, 50, 10\rangle$	124,141.39	203.79	225.54	56,685.84

Table 2: Nodes per second averaged over 100 seconds of pigeonhole instances where $n = 100$

4.5.2 The Anti-Chain Problem

In our second experiment we consider the anti-chain problem, defined below.

Definition 15. *An **anti-chain** is a set S of multisets where $\forall\{x, y\} \subseteq S.\ x \not\subseteq y \wedge y \not\subseteq x$.*

The $\langle n, l, d\rangle$ instance of anti-chain finds a set of n multisets with cardinality l drawn from d elements in total, satisfying the constraint of Definition 15. We model this as a CSP using n arrays of variables, denoted $M_1, \ldots, M_n$, each containing l variables with domain $\{0, \ldots, d-1\}$ and the constraints $\forall i \neq j \in \{1, \ldots, n\}.\ \exists k \in \{1, \ldots, n\}.\ M_i[k] < M_j[k]$.

Each variable $M_i[v]$ represents the number of occurrences of value v in multiset i, up to a maximum of $d - 1$. Each pair of rows M_i and M_j differ in at least two places: in one position k, $M_i[k] < M_j[k]$ and in another position p, $M_i[p] > M_j[p]$. This ensures that neither multiset contains the other.

Similarly to the generalised pigeon-hole problem, we consider 4 implementations of the constraint $\exists i.\ M[i] < N[i]$ for arrays M and N.

Watched OR: Implemented as a Watched OR.

Element: Introduce variables i with domain $\{0, \ldots, l-1\}$ and m and n each with domain $\{0, \ldots, d-1\}$. Impose the three constraints $M[i] = m, N[i] = n$ and $m < n$.

Sum: Introduce a new array of Boolean variables b of length l and impose the set of constraints $\forall i.\ (M[i] < N[i]) \leftrightarrow b[i]$ and also $\sum(b_{ij}) \geq 1$.

Watched Sum: The same model as **Sum**, except the constraint $(\sum b) \geq 1$ is replaced by a SAT clause $b[1] \vee \ldots \vee b[l]$ implemented with movable triggers.

$\langle n, l, d\rangle$	**Element**		**Watched OR**	
	Time	Nodes	Time	Nodes
$\langle 11, 4, 3\rangle$	3.83	142,674	0.64	77,117
$\langle 12, 4, 3\rangle$	77.40	3,030,555	9.68	2,189,034
$\langle 13, 4, 3\rangle$	4,488.45	166,888,355	416.89	95,301,659
$\langle 14, 4, 3\rangle$	4,931.10	166,888,372	444.87	95,301,661
$\langle 9, 4, 10\rangle$	3.76	90,678	0.34	12,349
$\langle 10, 4, 10\rangle$	28.31	636,635	0.73	75,807
$\langle 11, 4, 10\rangle$	171.82	3,340,225	3.16	399,997
$\langle 12, 4, 10\rangle$	775.36	12,311,354	15.68	1,815,755
$\langle 11, 5, 2\rangle$	735.76	47,602,427	188.30	47,602,427

Table 3: Search size for finding the first solution to the antichain problem

$\langle n, l, d\rangle$	**Element**			**Watched OR**		
	Time	Nodes	Solutions	Time	Nodes	Solutions
$\langle 2, 4, 3\rangle$	0.02	18,628	8,748	0.00	8,099	4,050
$\langle 3, 4, 3\rangle$	3.62	2,855,281	1,269,108	0.16	288,377	144,150
$\langle 4, 4, 3\rangle$	814.42	561,666,863	240,375,312	4.94	7,657,223	3,823,200
$\langle 3, 6, 2\rangle$	2.79	3,102,719	1,551,360	0.10	167,999	84,000
$\langle 3, 7, 2\rangle$	61.05	70,533,119	35,266,560	1.04	1,845,143	922,572

Table 4: Finding all solutions for instances of the antichain problem

We did not construct a custom propagator for this experiment because it takes considerable effort and we are concerned with generic algorithms.

Similarly to the previous experiment, the **Watched OR** , **Sum** and **Watched Sum** models all enforce the equivalent of GAC on the original expression, and **Element** does not.

Once again, we will consider the **Element** model separately, as we must compare time, rather than just nodes per second. In each of these experiments, we search for only the first solution and results are given in Table 3.

These results are much closer than those in the pigeon hole problem. On some instances, such as $\langle 11, 5, 2\rangle$, the **Element** model even achieves the same sized search as **Watched OR**. However, **Element** was slower in terms of nodes per second on all the instances we considered. Furthermore, **Element** sometimes exhibits a much larger number of solutions. Table 4 shows the results of finding all solutions to a small set of problems. The number of solutions found by the **Watched OR** model is the correct number of solutions, the **Element** duplicates some of these solutions multiple times, due to the fact its auxiliary variables can take multiple values for each solution to the problem. This shows once again the limitation of the **Element** model in practice.

To compare the other three models we consider how many nodes per second the particular model can solve, averaged over the first 100 seconds of search. In both cases we consider solving the anti-chain problem on 100 arrays ($n = 100$) of varying length

$\langle n, l, d \rangle$	Watched OR	Sum	Watched Sum
$\langle 100, 5, 2 \rangle$	22,351.38	727.77	984.32
$\langle 100, 10, 2 \rangle$	13,503.04	598.67	552.10
$\langle 100, 20, 2 \rangle$	8,812.30	466.01	640.04
$\langle 100, 30, 2 \rangle$	6,564.40	492.44	386.65
$\langle 100, 40, 2 \rangle$	6,426.52	434.64	374.49
$\langle 100, 50, 2 \rangle$	4,870.18	451.68	343.52
$\langle 100, 5, 10 \rangle$	344.69	42.23	32.89
$\langle 100, 10, 10 \rangle$	382.78	34.08	34.95
$\langle 100, 20, 10 \rangle$	385.45	39.59	37.25
$\langle 100, 30, 10 \rangle$	413.34	42.92	41.85
$\langle 100, 40, 10 \rangle$	506.79	56.50	54.96
$\langle 100, 50, 10 \rangle$	672.59	67.06	69.39

Table 5: Nodes per second achieved on antichain instances

and domain size.

A number of conclusions can be drawn from the results of this experiment, given in Table 5. First of all, our algorithm performs well compared to **Sum** on short vectors, but performance decreases as the length increases. For example with Boolean domains for length 5 arrays our algorithm is around 31 times faster, decreasing to 11 times at length 50. We note that for larger domains the nodes per second increases as the problem size increases. This is in common with the pigeon-hole problem, and is caused by a decrease in the proportion of variables with a movable trigger on them.

This experiment partially supports the hypothesis that both constraint trees and movable triggers are required to efficiently propagate OR. However we do not have an algorithm using static triggers with constraint trees, so we have not fully explored the space.

4.5.3 The Hamming Codes Problem

In this section we consider Hamming codes (see [19]), defined below.

Definition 16. *The $\langle n, l, d, s \rangle$ instance of the Hamming problem is to find a set of n codewords of length l with alphabet $\{1 \dots d\}$, where each pair of codewords differ in at least s positions.*

This is modelled as follows. We have n arrays of integers, named $M_1, \dots, M_n$, each of length l and domain $\{1, \dots, d\}$ with the following Hamming distance constraints: $\forall \{i, j\} \subseteq \{1, \dots, n\}$. $\left(\sum_{k \in \{1,\dots,l\}} M_i[k] \neq M_j[k]\right) \geq s$. We compare 3 representations of the constraint $\left(\sum_{i \in \{1,\dots,l\}} M[i] \neq N[i]\right) \geq s$.

Watched ATLEASTK: Directly represented as a Watched ATLEASTK, the algorithm described in Section 4.2.

Distance (s)	Watched AtLeastK	Sum	Watched Sum
49	12,660.01	10,932.19	1,746.42
45	55,012.06	50,508.98	4,446.96
40	53,005.73	52,200.72	9,491.13
30	159,528.97	54,943.08	19,336.15
20	184,247.49	54,770.75	30,900.58
10	76,531.80	20,405.95	13,281.60
5	1,502,509.71	1,613.26	1,639.26
3	1,733,255.03	1,226.18	1,239.55
2	1,726,415.28	1,166.95	1,166.50

Table 6: Nodes per second achieved on Hamming instances

Sum: Introduce an array of auxiliary Boolean variables $b[l]$ and add the set of constraints $\forall i \in \{1, \ldots, l\}. (M[i] \neq N[i]) \leftrightarrow b[i]$. Then impose $\sum b \geq s$.

Watched Sum: The same model as **Sum**, except the constraint $\sum b \geq s$ is replaced by a watched sum constraint.

For this problem we do not attempt to give an **Element** model, because preliminary experiments showed that the performance was so poor it was impossible to usefully compare it to any of the other models. All three models enforce GAC, because the child constraints of the AtLeastK do not share variables.

We experimented with the Hamming codes problem where $n = l = 50$ and $d = 2$, and the Hamming distance s is varied. The results are presented in Table 6. As stated in Section 4.2, we expect the Watched AtLeastK algorithm to be most efficient when k is small (where $k = s$ here). This is supported by Table 6, which shows Watched AtLeastK performing much better at low values of s than high values. However Watched AtLeastK dominates **Sum** and **Watched Sum** at all values of s. At $s = 49$, Watched AtLeastK will watch all child constraints, so there is little scope for it to improve on **Sum**.

In summary, this experiment provides some evidence that the gains from using constraint trees and movable triggers apply to AtLeastK as well as Or.

4.5.4 The Supertree Problem

The supertree problem [20] is that of transforming an input set of rooted bifurcating trees (*species trees*), describing the evolutionary history of a set of species, into an output tree respecting all the relationships in the input. Various CP models have been created to solve this problem, here we will use the model of [21] as well as the optimisation model of [22]. Both consist almost entirely of constraints of the form $(a \leq b = c) \vee (b \leq a = c) \vee (c \leq a = b) \vee (a = b = c)$. The standard model requires all such constraints to be satisfied, while the optimisation model maximises the number that are satisfied.

This can be modelled directly as $\text{Or}(\text{And}(a \leq b, b = c), \text{And}(b \leq a, a = c), \text{And}(c \leq a, a = b), \text{And}(a = b, a = c, b = c))$ using Watched And and Watched

Instance	Nodes	Watched time	Sum time	Saving
AB	58	0.32	0.42	22.12%
AD	97	0.45	0.79	42.24%
AF	66	0.32	0.40	20.60%
AG	152	0.41	0.73	43.87%
BD	72	0.54	0.72	24.65%
BF	27	0.33	0.43	24.36%
BG	78	0.47	0.73	35.12%
CD	53	0.58	1.14	48.50%
CF	30	0.38	0.53	29.14%
DF	81	0.94	1.07	12.76%

Table 7: Experimental data for solvable supertree instances

Instance	Best sol found	Watched time	Sum time	Saving
AC	48 cons satisfied	3,063.09	11,146.44	72.52%
BC	27	3,591.10	22,235.87	83.85%
CG	13	1,264.82	2,360.06	46.41%
DG	43	3,766.26	8,404.40	55.19%

Table 8: Experimental data for unsolvable supertree instances

OR. Note that this modelling does not require any auxiliary variables. The conjuncts and disjuncts share variables, so GAC may not be enforced by the Watched AND and OR propagators.

We compare this to the **Sum** model. We have already described how OR is handled using sums (Section 4.5.1). To represent AND, we reify each conjunct, and then use a sum constraint to represent the conjunction. This encoding uses auxiliary variables and enforces *the same* level of consistency as the above Watched OR and AND encoding.

We use all instances from Moore and Prosser [22] that have two input trees and are small enough to load. (The model takes cubic space and the larger instances exceeded 2GB RAM.) These are partitioned into ten solvable instances and four instances where input trees contain conflicting information (e.g. tree 1 says that a and b are closer relatives to each other than to c, whereas tree 2 says that a and c are closest). The standard model is used for the solvable instances, and the optimization model for the unsolvable ones.

Table 7 shows that the watched model is significantly faster than **Sum** for the ten solvable instances. These times do not include time to load instances, however load times are larger for **Sum** because it is less concise. Table 8 presents results for the unsolvable instances. We ran these instances to 2,000,000 nodes and again the results are in favour of the watched model. Using a profiler we discovered that the speedups are due to an increase in propagation speed; the reduced cost of creating, setting and backtracking the additional auxiliary variables has an insignificant effect in this case.

In summary, this final experiment shows that the Watched OR algorithm can be valuable when combined with another parent constraint.

5 Reification

The *reification* of a constraint C produces another constraint C_r, such that C_r has an extra Boolean variable r in its scope, and (in any solution) r is set to true iff the original constraint C is satisfied.

$$C_r \stackrel{def}{\equiv} r \Leftrightarrow C$$

Constraints can be combined in arbitrary ways using reification. For example, consider the exclusive-or of a set of constraints, as follows.

$$C_1 \oplus C_2 \oplus \cdots \oplus C_n$$

An odd number of these constraints must be satisfied in any solution. It is straightforward to represent this structure with reification. The constraints $C_1 \dots C_n$ are each reified, creating extra variables $r_1 \dots r_n$. These are added using a sum constraint, and the total variable is constrained to be odd.

Previous work on generic reification (Section 2.4) has been limited in one of two ways: the method cannot make use of efficient global propagators such as Régin's AllDifferent [14] (e.g. indexicals and propia [13, 15]); or GAC propagation is not achieved [16, 8]. Our methods overcome both these limitations, at the cost of requiring propagators for both C and $\neg C$.

In this section we describe two ways to propagate reified constraints, and compare them empirically. The first method uses only static triggers. The second method uses movable triggers, and is more complex, but it overcomes some of the apparent disadvantages of the first method.

We also investigate another form of reification, which we call *reifyimply*, where the reification variable implies the constraint, as follows.

$$C_{ri} \stackrel{def}{\equiv} r \Rightarrow C$$

Again we describe an algorithm based on checking and a movable trigger algorithm to propagate reifyimplied constraints.

5.1 Theoretical Analysis

Theorem 17 provides a simple algorithm which achieves GAC propagation for $r \Leftrightarrow C$, given a GAC propagator for both C and $\neg C$. We shall consider two different ways of making this algorithm more efficient, using incrementality. In general the propagators for C and $\neg C$ will be very different and can have very different complexities. Lemma 18 shows that the propagator for $r \Leftrightarrow C$ is tractable if and only if the propagators for both C and $\neg C$ are tractable.

Theorem 17. *The following Algorithm 3 is a GAC propagation algorithm for $r \Leftrightarrow C$ for Boolean variable r and any constraint C, assuming r is not in the scope of C and that the propagators for C and $\neg C$ achieve GAC propagation.*

```
Input: r, C
if Domain(r) = {TRUE, FALSE} then
    if There is no satisfying assignment to C then
        r ≠ TRUE
    end
    if There is no satisfying assignment to ¬C then
        r ≠ FALSE
    end
end
if Domain(r) = {TRUE} then
    Propagate(C)
else
    if Domain(r) = {FALSE} then
        Propagate(¬C)
    end
end
```

***Algorithm 3**: GAC propagation algorithm for reify*

Proof. Consider the following cases upon entering the algorithm:

1. $Domain(r) = \{\textbf{TRUE}, \textbf{FALSE}\}$**:** In this case, we check if the values in r are supported. This requires finding both an assignment to $\mathcal{X}_C$ which satisfies C, and an assignment which does not satisfy C. If either value is unsupported it is removed, and the algorithm continues with case 2 below.

   If neither value of r is removed then every value in the domain of every variable in the scope of C is supported, by either C or $\neg C$. Any assignment to the variables in C can be extended to a satisfying assignment to $r \Leftrightarrow C$ by adding either $r = \text{TRUE}$ or $r = \text{FALSE}$, depending on whether the assignment satisfies C or $\neg C$.

2. $Domain(r)$ **contains a single value:** In this case, if the domain of r is $\{\text{TRUE}\}$, $r \Leftrightarrow C$ is exactly equivalent to C, and if the domain of r is $\{\text{FALSE}\}$, the constraint is equivalent to $\neg C$. □

Lemma 18. *GAC($r \Leftrightarrow C$) is NP-hard if and only if at least one of GAC(C) and GAC($\neg C$) is.*

Proof. Running GAC(C) on a subdomain list removes all domain values if and only if there is no satisfying assignment for C. Therefore, Theorem 17 demonstrates how to implement GAC($r \Leftrightarrow C$) using at most one invocation of GAC(C) and at most one invocation of GAC($\neg C$). Therefore GAC($r \Leftrightarrow C$) is polynomial time if both GAC(C) and GAC($\neg C$) are. By assigning r to TRUE or FALSE, we can see that GAC($r \Leftrightarrow C$) must be at least as hard as both GAC(C) and GAC($\neg C$). □

In this paper reification is implemented as a constraint tree with two child constraints, C and $\neg C$. This raises the issue of shared variables among child constraints, as discussed in Section 2.4.2. However, the constraint tree propagator implements Algorithm 3, and therefore enforces GAC despite the shared variables.

Theorem 19 presents a basic algorithm for implementing the constraint $r \Rightarrow C$. We will improve this basic algorithm using incrementality.

Theorem 19. *The following Algorithm 4 is a GAC propagation algorithm for $r \Rightarrow C$ for Boolean variable r and any constraint C, assuming r is not in the scope of C and the propagator for C achieves GAC.*

Input: r, C
if $Domain(r) = \{\text{TRUE}, \text{FALSE}\}$ **then**
  if *There is no satisfying assignment to C* **then**
    $r \neq \text{TRUE}$;
  end
else
  if $Domain(r) = \{\text{TRUE}\}$ **then**
    Propagate(C)
  end
end

Algorithm 4: *GAC propagation algorithm for reifyimply*

Proof. This proof follows the cases in the algorithm:

1. $Domain(r) = \{$**TRUE**, **FALSE**$\}$**:** In this case, every value in the domain of every variable in the scope of C is supported, as an assignment which contains $r = \text{FALSE}$ satisfies the constraint. Therefore the only value which could possibly be eliminated is $r = \text{TRUE}$. This value is allowed if and only if there exists an assignment to $\mathcal{X}_C$ which satisfies C.

2. $Domain(r)$ **contains a single value:** In this case, if the domain of r is $\{\text{TRUE}\}$, the constraint is exactly equivalent to just C, and if the domain of r is $\{\text{FALSE}\}$, any assignment satisfies the constraint so no pruning can occur. □

5.2 Algorithms for Reification and Reifyimply

The following algorithms for $r \Leftrightarrow C$ and $r \Rightarrow C$ have some features in common. They all have a phase for checking entailment/disentailment of C, so that r can be set when necessary (the *watching* or *checking* phase). They all have a phase for propagating C (or $\neg C$) when that is necessary (the *propagation* phase). The movable trigger algorithms also have a setup phase where movable triggers are placed for the first time.

When describing the algorithms, C is described as a *child* constraint object, with methods for propagation, checking disentailment (*checkUnsat*) and a satisfying set generator. Checking for disentailment is equivalent to checking if a satisfying set generator would return FAIL. This means it can often be implemented more efficiently. Full reification also has $\neg C$ as a child. Child constraints do not receive trigger events unless they are passed through by the parent.

5.3 Watched Reification

First we describe implementing reification using movable triggers. Following this, we will show three simple modifications of this algorithm. In this scheme, both the positive and negative child constraints must implement a satisfying set generator. Watched reification has three phases, described below. There are three sets of triggers: triggers required by the child constraints; the static trigger on r; movable triggers placed in phases 1 and 2 to watch satisfying sets.

Setup Phase: If r is assigned, move to the propagation phase. Otherwise, call the satisfying set generator for both child constraints. If either child returns FAIL, then it is disentailed. Set r appropriately and move to the propagation phase. Otherwise, place static triggers on r and movable triggers on both satisfying sets and move to the watching phase.

Watching Phase: If r is assigned, move to the propagation phase. If a domain value being watched is removed, then determine which child it belongs to, and call the satisfying set generator again for the child. If it returns FAIL, set r appropriately and move to the propagation phase. If it returns a satisfying set, place movable triggers on it and remain in this phase.

Propagation Phase: If $r = 1$ then propagate the positive constraint, otherwise propagate the negative constraint. Trigger events for the appropriate child constraint are passed through.

Since movable triggers are not backtracked, it is possible to receive stale trigger events from movable triggers which were placed in a different phase. Therefore in the watching and propagation phases, some trigger events must be ignored or otherwise handled specially. These are listed below.

Watching Phase: Trigger events from the propagation phase may be received in this phase; in this case the movable trigger is removed and the event is ignored. Any trigger events belonging to child constraints are ignored.

Propagation Phase: When propagating one child constraint, trigger events for the other child are ignored. Movable trigger events from setup and watching phases are ignored.

Notice that movable triggers from the setup and watching phases are not removed in the propagation phase. When backtracking into the watching phase, there is no opportunity to place movable triggers, but the previous set are still be present so there is no need to replace them.

The setup phase only occurs when the propagator is first invoked. The other two phases occur during search, and we use one backtracking Boolean to indicate which phase the algorithm is in. This algorithm does not make use of the fact that disentailment of C implies entailment of $\neg C$, and therefore can perform unneccesary propagation of entailed child constraints. We leave this for future work.

5.3.1 Reifyimply

We implemented watched reifyimply, using the abstract Algorithm 4. As Algorithm 4 is a subset of Algorithm 3, we did this by taking a subset of the concrete algorithm described in Section 5.3 above. This required removing the child constraint $\neg C$, as it is not necessary to check disentailment of, or propagate, $\neg C$. Also, it is only necessary to trigger when r is assigned 1, as no propagation occurs when r is assigned 0.

5.4 Static Reification

We implement a static variant of both reify and reifyimply. These use a disentailment checker instead of a combination of a satisfying set generator and movable triggers to detect when a constraint is disentailed. Static reification requires both the positive and negative child constraints have a *checkUnsat* method which checks if the constraint is disentailed. Before search begins, the (static) triggers of both the positive and negative constraints are placed on the variables, along with a trigger on the reification variable.[5]

5.5 Empirical comparison of reification algorithms

In this section we give an empirical comparison of reify and reifyimply, in their watched and static forms, using a range of realistic benchmark problems.

Notice that checkUnsat (CU) in static reification, and satisfying set generators (SSG) in watched reification perform similar tasks. Both determine whether a constraint is disentailed. Satisfying set generators additionally return a satisfying set of literals when the constraint is not disentailed. For all reified or reifyimplied constraints in the benchmarks, the two functions are equivalent for determining disentailment. Hence, static and watched algorithms provide the same level of consistency, and the solver explores the same number of search nodes for all benchmarks.

One metric we use to compare static and watched algorithms is the number of calls made to CU and SSG. Consider a hypothetical solver which only offers triggers (static or watched) on individual domain values. CU must have static triggers on any value which may be important at any time during search. SSG is able to place watches during search. In this solver SSG cannot be called more times than CU. In most cases, this carries through to Minion, however Minion has assignment triggers which are not available to SSG. For movable triggers to have any potential, the number of calls to SSG must be substantially fewer, since the cost of calling it is somewhat higher and there is the additional overhead of placing movable triggers.

The methodology and hardware used for the following experiments was the same as for those described in §4.5.

5.5.1 Steel Mill Slab Design

Our first benchmark consists of instances of the steel mill slab design problem [23]. This is a well-known optimisation problem involving assigning orders to a steel mill to slabs, minimising the total waste. Our instances include reifyimplied lex ordering

[5] Our implementation of static reification does not allow children to use movable triggers.

Instance	Watched		Static		Overall
	Time	Calls to SSG	Time	Calls to CU	Winner
40	2,120.02	151	2,180.97	1,350,922,599	W by 2.87%
50	2,329.73	131	2,362.71	758,088,075	W by 1.42%
60	2,839.90	156	2,884.10	1,193,581,857	W by 1.56%
70	3,542.02	246	3,619.98	1,589,486,539	W by 2.20%
80	4,488.40	282	4,622.78	2,525,193,986	W by 2.99%
90	5,390.12	320	5,248.09	2,596,608,279	S by 2.63%

Table 9: Times and call counts for steelmill instances

constraints on rows of a 0/1 matrix, these constraints break symmetry on the rows and are reifyimplied so that they can be switched off when a row (corresponding to a slab) is not needed to fulfil the set of orders.

Our evaluation on these instances exhibits solid results in favour of watched reifyimply. Table 9 shows an exceptional decrease in calls to SSG compared to CU, for watched versus static reifyimply, running the instances up to 100,000,000 nodes. Here billions of calls are being made to CU compared to hundreds for SSG. In fact, after the first 100 nodes of search in all these examples, the movable triggers are hardly ever moved. Instance 90 is typical: during the first 100 nodes, SSG is called 260 times; at 10,000 nodes it has been called 301 times; and at 1,000,000 nodes it has been called 315 times. For the same instance CU is being called over 60 times per node on average up to 1,000,000 nodes. This dramatic improvement is due to the movable triggers being very rarely triggered in the watched variant, whereas for the static variant the bound triggers are being woken up frequently even when the constraint remains satisfiable. SSG needs to place movable triggers on just two values in the scope of the lexleq needed to ensure it remains satisfiable, whereas CU has bound triggers on all the variables in the scope of the constraint.

Table 9 shows that this improvement in calls translates to an improvement in solution time. This improvement is relatively small in absolute terms, but this is because most of the time is spent propagating other constraints besides reifyimply. With the aid of a profiler, we have discovered that, on benchmark 90, the average call to SSG for the lexicographic ordering constraint consumes 2695 CPU instructions whereas the average call to CU consumes just 54. These statistics give an impression that the SSG movable triggers must be triggered substantially less often than the static triggers to justify the cost, in this case more than 50 times less often (since there is an additional overhead of placing dynamic triggers on the literals).

5.5.2 Blackhole Solitaire

Blackhole solitaire [24] is a single-player card game. The initial layout is 17 stacks of 3 cards, with all cards visible. There is one special stack, containing only the ace of spades initially, named the black hole. Cards are moved from the top of a stack onto the black hole, and the game is completed when all 51 cards have been moved onto the black hole. The card moved must be adjacent to (but not the same as) the previous

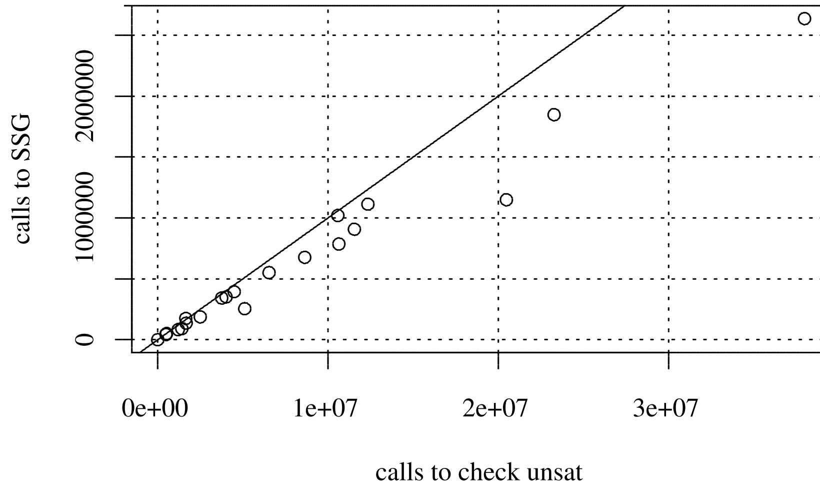


Figure 3: Comparison of calls to SSG and CU for blackhole problems

card on the black hole, regardless of suit, where adjacency wraps around (i.e. king is adjacent to ace). A solution is a sequence of 51 valid moves.

Our model of blackhole solitaire contains reifyimplied less-than constraints ($r \Rightarrow x_1 < x_2$). The less-than constraint places two static triggers, one on the lower bound of x_1 and the other on the upper bound of x_2. SSG always returns two movable triggers, the lower bound of x_1 and the upper bound of x_2. When bounds are restored on backtracking, the movable triggers are no longer on the bounds. This effect allows SSG to be called many fewer times than CU on these benchmarks. The model also contains reified less-than and sum-greater constraints, which were propagated statically in both cases, so as not to influence the results.

As shown in Figure 3 the total number of calls to SSG for all constraints is much smaller than the number of calls to CU for each instance of blackhole we tried. The black line on the plot is the line $y = x/10$, or the "10 times better line", since all points beneath the line use at least 10 times more calls to CU than SSG, for static and watched reifyimply respectively. Using a profiler, we have discovered that the mean number of CPU instructions in a call to SSG was 54 versus 9 instructions per call to CU, meaning that the ratio of CU to SSG would have to be more than 6 for dynamic reify-imply to have a chance of winning. This does not take into account the time to additionally place the watches, and so Figure 4 shows that even a ratio of 10 is not sufficient, as the static algorithm is slightly faster on this benchmark.

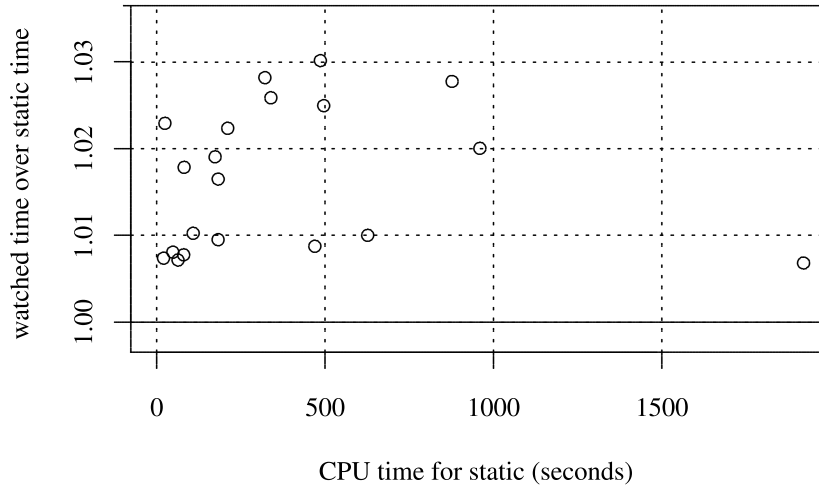


Figure 4: Comparison of time spent on blackhole problems

5.5.3 Contrived benchmark

We use a reified allDifferent constraint in a contrived problem intended to demonstrate the potential of watched reification. We expect that watched reification will perform well if the movable triggers can settle on values which are never (or only rarely) removed. This effect was observed for watched reifyimply, on the steel mill slab design problem.

Problem instances can be generated for any positive integer k, and consist of two k-vectors X and Y with domains $\{1, \ldots, k\}$. The constraints are as follows: $\forall i \in \{1 \ldots k\} : (2X[i]) \neq Y[i]; X[k-1] \neq X[k]; X[k-1] = X[k]$ and $r \Leftrightarrow$ allDifferent(Y).

The allDifferent constraint uses a GAC algorithm [25], and maintains a matching from variables to distinct values. SSG for the positive constraint returns a k-matching if one exists, hence there is one movable trigger for each variable in X. For the static reify, CU is called for any domain change. CU is very similar to SSG, it maintains a maximal matching using the same algorithm as SSG.

The negative constraint waits until all variables are assigned, then checks the assignment[6]. SSG for the negative constraint places two movable triggers on different values of an unassigned variable, if possible. If all variables are assigned, SSG checks if the constraint is disentailed. CU requires an assignment trigger on each variable.

The variable ordering is X in index order, values are branched in ascending order. $X[k]$ cannot be consistently assigned, and there is no restriction on the rest of X, so

[6]The standard implementation in Minion 0.10 is a Watched OR of equal constraints on all pairs of variables. Unfortunately, Watched OR is incompatible with the static reification algorithm, so for this experiment the Watched OR was replaced with an assignment checker.

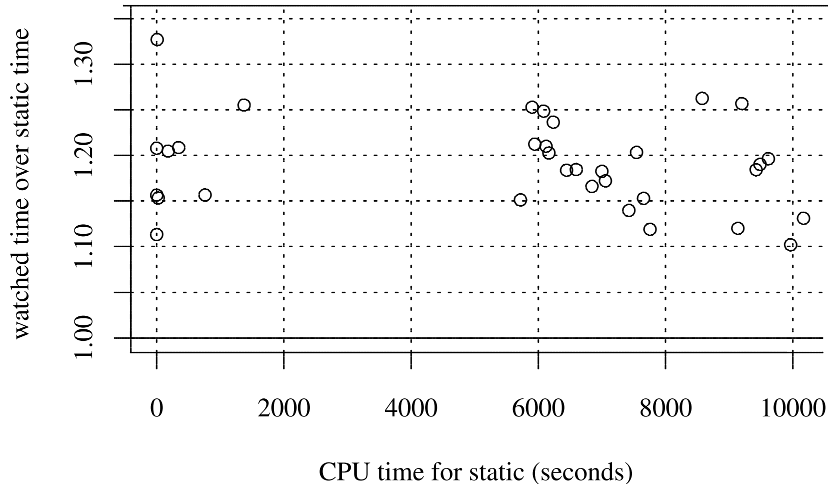


Figure 5: Comparison of time spent on English peg solitare instances

the solver explores k^{k-1} assignments of $X[1 \dots k-1]$. Whenever a variable $X[i]$ is set to j, $2j$ is removed from $Y[i]$ by the not-equal constraint. Therefore odd values in Y are never removed, and movable triggers may settle on them.

We ran instance $k = 20$ with a node limit of 10,000,000. Watched reify made 2509 calls to SSG, compared to 10526315 calls to CU. With static reify, Minion took 50.82s, and with watched reify it took 50.16s. Using the *callgrind* profiler (and a node limit of 500,000), we found that Minion uses 6.60 bn CPU instructions with static reify and 6.42 bn with watched reify. The static reify propagator alone uses 193m instructions, compared to 7.90m for the watched reify propagator. This clearly shows that most of the cost is outside the reification, and that watched reify is performing much better than the static variant, as we would expect from the call counts.

5.5.4 English Peg Solitaire

Finally we consider the game of English peg solitaire [26], which is played with 32 pegs placed in a board with 33 holes. Pegs are removed by hopping moves (similar to checkers/draughts) until a goal state is reached or no moves are possible. We use model C of Jefferson et al [26], slightly adapted to suit Minion rather than ILOG Solver. These benchmarks contain a large number of reified sum constraints. The constraints state that a sum of Boolean variables is 1 or more. The length of the sum ranges from 1 to 8 variables.

We used 33 instances with different goals. All instances are run to a node limit of 10,000,000. Figure 6 shows that, on these instances, the number of calls to SSG by watched reification is usually between a half and third of the calls to CU for static

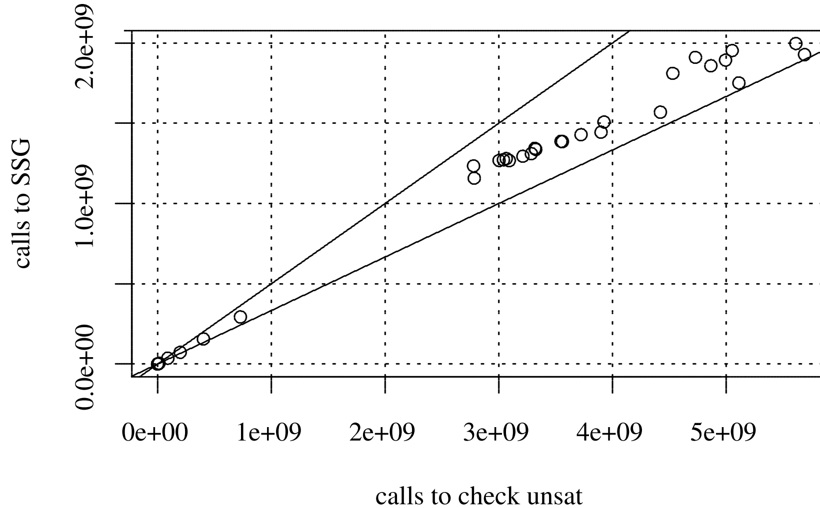


Figure 6: Comparison of calls to SSG and CU for English peg solitaire instances

reification. However, Figure 5 shows that static reification is faster for all instances.

We used the profiler *callgrind* with instance `solitaire_benchmark_6` (which takes 40s with static reification). Minion uses 71.2 billion CPU instructions with static reification, and 78.1 with watched. Static reify alone uses 21.8 bn, and watched reify uses 26.5, an increase of 22%[7].

5.5.5 Conclusion to empirical comparison

The results of our experiments are not conclusive, demonstrating that different implementations perform better on different constraints and problems. In all cases, we have shown the potential of a movable triggers approach, by demonstrating that the SSG function is called much less often than CU. On the other hand, static reification (and reifyimply) is simple and fast, and in many cases it is faster than the watched variant.

6 Conclusion

In this paper we have explored possibilities for implementing logical connectives in a constraint solver, with the overall hypothesis that movable triggers and constraint trees together are invaluable. These two solver features are combined with satisfying set generators, which provide an efficient way of checking the satisfiability of a constraint.

[7]Changing the reification algorithm changes the propagation order and affects other constraints. In this case, the difference for reify alone is 4.8 bn and for the whole solver it is 6.9 bn.

First we focussed on ATLEASTK, OR and AND of arbitrary constraints. The ubiquitous way of modelling these in CP is by reifying the constraints, and applying a sum-$\geq k$ constraint (or equivalent) to the reification variables. With this approach, the solver is required to propagate all reified constraints at all times. By contrast, the Watched ATLEASTK algorithm we present has at most $k + 1$ active constraints at any time — all others have zero cost. Using this approach on Hamming codes we were able to demonstrate a 2,000 times speedup on some instances compared to reification.

We also presented Watched OR, a specialisation of Watched ATLEASTK. In our evaluation we observed that Watched OR can be over 10,000 times faster than reification, and is consistently much faster on all problems we tested.

By implementing satisfying set generators for Watched ATLEASTK, OR and AND, these parent constraints can be arbitrarily nested, giving a rich language for logical expressions. We hope to extend this work to other logical connectives, and also to achieve GAC in the case where child constraints share variables, while maintaining high performance.

Secondly, we investigated two ways of implementing both reification and reifyimply for any constraint. We described simple algorithms which use static triggers, and more sophisticated algorithms which make use of movable triggers to reduce the number of constraint checks. In our experiments, the results were mixed. In some cases, the simple static algorithms were faster, and in others the watched algorithms paid their additional overhead and were more efficient.

The common thread through this paper is that movable triggers, satisfying sets and constraint trees together allow simple, efficient implementation of logical connectives of constraints. Once a constraint has a satisfying set generator (which is usually much simpler than its propagation function), it can be used in Watched OR and other parent constraints, and it can be reified and reifyimplied. This makes a simple, general and compelling framework for implementing logical connectives.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments about an earlier version of this paper. This work was funded by EPSRC research grant numbers EP/C523229/1 (Jefferson), EP/H004092/1 (Jefferson, Nightingale, Petrie), EP/E030394/1 (Moore, Nightingale), and a Royal Society Dorothy Hodgkin Fellowship (Petrie).

References

[1] Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In Benhamou, F., ed.: CP. Volume 4204 of Lecture Notes in Computer Science., Springer (2006) 182–197

[2] Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)

[3] Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier (2006)

[4] Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: ECAI. Volume 141 of Frontiers in Artificial Intelligence and Applications., IOS Press (2006) 98–102

[5] Brand, S., Yap, R.H.C.: Towards "propagation = logic + control". In Etalle, S., Truszczynski, M., eds.: ICLP. Volume 4079 of Lecture Notes in Computer Science., Springer (2006) 102–116

[6] Müller, T., Würtz, J.: Constructive disjunction in Oz. In: Workshop Logische Programmierung (WLP). Volume 270 of GMD-Studien., Gesellschaft für Mathematik und Datenverarbeitung MBH (1995) 113–122

[7] Würtz, J., Müller, T.: Constructive disjunction revisited. In Görz, G., Hölldobler, S., eds.: German Conference on Artificial Intelligence (KI) 1996. Volume 1137 of LNCS., Springer (1996) 377–386

[8] Lagerkvist, M.Z., Schulte, C.: Propagator groups. In Gent, I.P., ed.: CP. Volume 5732 of Lecture Notes in Computer Science., Springer (2009) 524–538

[9] Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI, Professional Book Center (2005) 35–40

[10] Lhomme, O.: An efficient filtering algorithm for disjunction of constraints. In Rossi, F., ed.: CP. Volume 2833 of Lecture Notes in Computer Science., Springer (2003) 904–908

[11] Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In Régin, J.C., Rueher, M., eds.: CPAIOR. Volume 3011 of Lecture Notes in Computer Science., Springer (2004) 209–224

[12] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: DAC '01: Proceedings of the 38th annual Design Automation Conference, New York, NY, USA, ACM (2001) 530–535

[13] Hentenryck, P.V., Saraswat, V., Deville, Y.: Constraint processing in cc(fd). Technical report, Brown University (1991)

[14] Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings 12th National Conference on Artificial Intelligence (AAAI 94). (1994) 362–367

[15] Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Harvey, W., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Novello, S., Perez, B., van Rossum, E., Schimpf, J., Shen, K., Tsahageas, P.A., de Villeneuve, D.H.: Eclipse user manual release 5.10 (2006) http://eclipse-clp.org/.

[16] Schulte, C.: Programming deep concurrent constraint combinators. In: Proceedings of Practical Aspects of Declarative Languages (PADL 2000). Volume 1753 of LNCS., Springer (2000) 215–229

[17] Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. Constraints **12**(2) (2007) 239–259

[18] Jefferson, C.: Representations in Constraint Programming. PhD thesis, University of York (2007)

[19] Colbourn, C.J., Dinitz, J.H.: Handbook of Combinatorial Designs, Second Edition (Discrete Mathematics and Its Applications). Chapman & Hall/CRC (2006)

[20] Daniel, P., Semple, C.: Supertree algorithms for nested taxa. In Bininda-Emonds, O., ed.: Phylogenetic Supertrees: Combining information to reveal the tree of life. Computational Biology Series Kluwer (2004) 151–171

[21] Gent, I.P., Prosser, P., Smith, B.M., Wei, W.: Supertree construction with constraint programming. In Rossi, F., ed.: CP. Volume 2833 of Lecture Notes in Computer Science., Springer (2003) 837–841

[22] Moore, N.C., Prosser, P.: The ultrametric constraint and its application to phylogenetics. Journal of Artificial Intelligence Research **32** (Aug 2008) 901–938

[23] Frisch, A.M., Miguel, I., Walsh, T.: Modelling a steel mill slab design problem. In: IJCAI-01 Workshop on Modelling and Solving Problems with Constraints. (2001) 39–45

[24] Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M.: Search in the patience game 'black hole'. AI Communications **20**(3) (2007) 211–226

[25] Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artificial Intelligence **172**(18) (2008) 1973–2000

[26] Jefferson, C., Miguel, A., Miguel, I., Tarim, A.: Modelling and solving english peg solitaire. Computers and Operations Research **33**(10) (2006) 2935–2959