Constraints (2009) 14:357–391
DOI 10.1007/s10601-008-9064-x

Propagation via lazy clause generation

Olga Ohrimenko · Peter J. Stuckey · Michael Codish

Published online: 13 January 2009

Abstract Finite domain propagation solvers effectively represent the possible values of variables by a set of choices which can be naturally modelled as Boolean variables. In this paper we describe how to mimic a finite domain propagation engine, by mapping propagators into clauses in a SAT solver. This immediately results in strong nogoods for finite domain propagation. But a naive static translation is impractical except in limited cases. We show how to convert propagators to lazy clause generators for a SAT solver. The resulting system introduces flexibility in modelling since variables are modelled dually in the propagation engine and the SAT solver, and we explore various approaches to the dual modelling. We show that the resulting system solves many finite domain problems significantly faster than other techniques.

Keywords Finite domain propagation · Boolean variables · SAT solver

1 Introduction

Propagation is an essential aspect of finite domain constraint solving which tackles hard combinatorial problems by interleaving search and restriction of the possible values of variables (propagation). The propagators that make up the core of a finite domain propagation engine represent trade-offs between the speed of inference of information versus the strength of the information inferred. Good propagators represent a good trade-off at least for some problem classes. The success of finite

This paper is an extension of results first published in [29, 30].

O. Ohrimenko · P. J. Stuckey
NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng.,
University of Melbourne, Melbourne, Australia

M. Codish (✉)
Department of Computer Science, Ben-Gurion University, Beersheba, Israel
e-mail: mcodish@cs.bgu.ac.il

domain propagation in solving hard combinatorial problems arises from these good trade-offs, and programmable search.

Propositional satisfiability (SAT) solvers are becoming remarkably powerful and there is an increasing number of papers which propose encoding hard combinatorial (finite domain) problems in SAT (e.g. [21, 33]). The success of modern SAT solvers is largely due to a combination of techniques including: watch literals, 1UIP nogoods and the VSIDS variable ordering heuristic [27].

In this paper we propose modelling combinatorial problems in SAT, not by modelling the constraints of the problem, but by modelling/mimicking the propagators used in a finite domain model of the problem. Variables are modelled in terms of the changes in domain that occur during the execution of propagation. We can then model the domain changing behaviour of propagators as clauses.

Encoding finite domain propagation can test the limits of SAT solvers. While modern SAT solvers can often handle problems with millions of clauses and hundreds of thousands of variables, many problems are difficult to encode into SAT without breaking these implicit limits. We propose a hybrid approach. Instead of introducing clauses representing propagators *a priori*, we execute the original (finite domain) propagators as lazy clause generators inside the SAT solver. Propagators introduce their propagation clauses precisely when they are able to trigger new unit propagation. The resulting hybrid combines the advantages of SAT solving, in particular powerful and efficient nogood learning and backjumping, with the advantages of finite domain propagation, simple and powerful modelling and specialized and efficient propagation of information.

Example 1 Consider propagation on the constraint $x + y = z$ where x, y and z range over values from -1000 to 1000. In a finite domain propagation engine, the propagator is a tiny and efficient piece of code. For example given that x and y can take values only less than or equal to 5, the propagator determines that z can take only values less than or equal to 10. The propagator is highly efficient and invoked whenever the domains of x, y or z change.

Encoding this simple constraint using clauses on the Boolean variables $[\![v \leqslant d]\!]$ where $v \in \{x, y, z\}$ and $d \in [-1000 .. 1000]$ involves over 2 million clauses, for example $\neg[\![x \leqslant 5]\!] \vee \neg[\![y \leqslant 5]\!] \vee [\![z \leqslant 10]\!]$, and $\neg[\![x \leqslant -500]\!] \vee \neg[\![y \leqslant -501]\!]$. This is a huge representation for such a simple constraint.[1]

In lazy clause generation, the domains of the integer variables are represented using Boolean variables, so the Boolean variable $[\![x \leqslant 5]\!]$ represents that x can take values only less than or equal to 5. Rather than build the representation of the constraint statically before execution it is built lazily during search. During search the propagator is run, and its results are converted to clauses which are added to the SAT solver. For example when $x \leqslant 5$ and $y \leqslant 5$ the propagator determines that $z \leqslant 10$, this becomes the clause $\neg[\![x \leqslant 5]\!] \vee \neg[\![y \leqslant 5]\!] \vee [\![z \leqslant 10]\!]$. This clause is added to the SAT solver. Since during any execution many less combinations of domains for x, y and z will occur than are actually possible, only very few of the more than 2 million clauses will end up in the SAT solver.

[1] Smaller representations based on logarithmic encodings exist but they do not propagate nearly as strongly.

The advantage of creating the clauses is that the SAT solver can then be used to direct search using its specialized search methods, and more importantly the SAT mechanisms for nogood learning and backjumping can be used. Moreover, note that clauses added by a propagator are globally true and never need to be retracted. For example, if $x \leqslant 5$ and $y \leqslant 5$ becomes untrue due to backtracking then the propagation clause $\neg[\![x \leqslant 5]\!] \vee \neg[\![y \leqslant 5]\!] \vee [\![z \leqslant 10]\!]$ simple no longer fires under unit propagation and $z \leqslant 10$ is no longer implied. Indeed the clause can propagate in other ways: give $x \leqslant 5$ and $z \geqslant 11$ (equivalently $\neg z \leqslant 10$) the SAT solver can determine that $y \geqslant 6$. And this is a true consequence of $x + y = z$.

We show that the lazy clause generation approach allows independence between the Boolean representation of integer variables and the propagators that act upon them. This representation independence leads to a new type of propagation: mixing bounds representation and domain propagators. The new propagator results in disjunctive propagation, where new information is created by propagation which is disjunctive in nature, even though the propagator was not disjunctive initially. Since the underlying SAT representation of propagation can represent disjunctive information efficiently, it allows us to create new "disjunctive propagators" from scratch.

We compare our hybrid solving approach to finite domain propagation-based solving and static modelling and solving using SAT on a number of benchmark problems. We show that our prototype implementation can significantly improve on the carefully engineered SAT and finite domain propagation solvers. We also illustrate how the separation of Boolean model of the problems from the style of propagation can lead to improved behaviour of the hybrid approach.

The contributions of this work are:

- A formalization of propagators in terms of clauses, and the correspondence result between using the propagators and SAT solving on the resulting clauses;
- Design of a hybrid system for implementing propagation-based solving with a SAT solver;
- Exploration of the modelling choices that arise from the hybrid system including novel "disjunctive propagators" which create clauses in the SAT solver earlier than standard propagators;
- The first system we are aware of that combines nogoods with bounds propagators; and
- Experimental evidence of the significant potential of the hybrid approach.

The remainder of this paper is organized as follows. In Section 2 we give our terminology for finite domain constraint solving and in Section 3 we give our terminology for SAT problems and unit propagation. In Section 4 we introduce atomic constraints and propagation rules as a way of understanding the pointwise behaviour of propagators. Then in Section 5 we show how we can represent propagators as clauses. In Section 6 we introduce the concept of lazy clause generation, where we lazily build a clausal representation of a propagator. In Section 7 we explore some of the modelling choices that arise from the dual viewpoint of variables as Boolean literals. In Section 8 we discuss issues in implementing the lazy clause generation approach, and in Section 9 we show the results of a number of experiments. We discuss related work in Section 10 and then conclude.

2 Propagation-based constraint solving

We consider a typed set of variables $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_S$ made up of *integer* variables, $\mathcal{V}_I$, and *sets of integers* variables, $\mathcal{V}_S$. We use lower case letters such as x and y for integer variables and upper case letters such as S and T for sets of integers variables. A *domain* D is a complete mapping from $\mathcal{V}$ to finite sets of integers, for the variables in $\mathcal{V}_I$, and to finite sets of finite sets of integers, for the variables in $\mathcal{V}_S$. We can understand a domain D as a formula $\wedge_{v\in\mathcal{V}}(v \in D(v))$ stating for each variable v that its value is in its domain.

Let D_1 and D_2 be domains and $V \subseteq \mathcal{V}$. We say that D_1 is *stronger* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$ and that D_1 and D_2 are *equivalent modulo* V, written $D_1 =_V D_2$, if $D_1(v) = D_2(v)$ for all $v \in V$. The *intersection* of D_1 and D_2, denoted $D_1 \sqcap D_2$, is the domain which maps every $v \in \mathcal{V}$ to $D_1(v) \cap D_2(v)$.

We use *range* notation: For integers l and u, $\left[\, l \,..\, u \,\right]$ denotes the set of integers $\{d \mid l \leqslant d \leqslant u\}$, while for sets of integers L and U, $[\, L \,..\, U \,]$ denotes the set of sets of integers $\{A \mid L \subseteq A \subseteq U\}$. A domain D is *convex* if $D(T)$ is a range for all $T \in \mathcal{V}_S$. We restrict attention to convex domains. We assume an *initial domain* D_{init} which is convex such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{init}$.

A *valuation* θ is a mapping of integer and set variables to correspondingly typed values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, S_1 \mapsto A_1, \ldots, S_m \mapsto A_m\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in vars(\theta)$.

A constraint is a restriction placed on the simultaneously allowed values for a set of variables. We define the *solutions* of a constraint c to be the set of valuations θ that make that constraint true, i.e. $solns(c) = \{\theta \mid (vars(\theta) = vars(c)) \ \wedge \ (\models \theta(c))\}$

We associate with every constraint c a set of *propagators*, $prop(c)$. A propagator $f \in prop(c)$ is a monotonically decreasing function on domains that is solution preserving for c. That is for all domains $D \sqsubseteq D_{init}$: $f(D) \sqsubseteq D$ and

$$\{\theta \in D \mid \theta \in solns(c)\} = \{\theta \in f(D) \mid \theta \in solns(c)\}.$$

This is a weak restriction since, for example, the identity mapping is a propagator for any constraint. In this paper we restrict ourselves to *set bounds propagators* that map convex domains to convex domains.

Example 2 A common propagator f for the constraint $x \neq y$ is

$$f(D)(v) = \begin{cases} D(v) - \{d\} & \text{if } (v = x \text{ and } D(y) = \{d\}) \text{ or} \\ & \quad (v = y \text{ and } D(x) = \{d\}) \\ D(v) & \text{if } (v = x \text{ and } |D(y)| > 1) \text{ or} \\ & \quad (v = y \text{ and } |D(x)| > 1) \text{ or} \\ & \quad (v \notin \{x, y\}) \end{cases}$$

Consider a domain D where $D(x) = \{3, 4, 5, 6\}$ and $D(y) = \{5\}$, then $f(D)(x) = \{3, 4, 6\}$ and $f(D)(y) = \{5\}$.

The *output* variables $output(f) \subseteq \mathcal{V}$ of a propagator f are the variables changed by the propagator: $v \in output(f)$ if $\exists D \sqsubseteq D_{init}$ such that $f(D)(v) \neq D(v)$. The *input* variables $input(f) \subseteq \mathcal{V}$ of a propagator f are the variables that can change the result of the propagator, that is the smallest subset $V \subseteq \mathcal{V}$ such that for each $D \sqsubseteq D_{init}$ and $D' \sqsubseteq D_{init}$: $D =_V D'$ implies that $f(D) \sqcap D' =_{output(f)} f(D') \sqcap D$. Only the input variables are useful in computing the application of the propagator to the domain.

Example 3 For the constraint $c \equiv x_1 + 1 \leqslant x_2$ the function f defined by $f(D)(x_1) = \{d \in D(x_1) \mid d \leqslant \max D(x_2) - 1\}$ and $f(D)(v) = D(v), v \neq x_1$ is a propagator for c. Its output variables are $\{x_1\}$ and its input variables are $\{x_2\}$. Let $D(x_1) = \{3, 4, 6, 8\}$ and $D(x_2) = \{1, 5\}$, then $f(D)(x_1) = \{3, 4\}$ and $f(D)(x_2) = \{1, 5\}$.

A *propagation solver* for a set of propagators F and current domain D, $solv(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in the resulting domain. $solv(F, D)$ is the weakest domain $D' \sqsubseteq D$ which is a fixpoint (i.e. $f(D') = D'$) for all $f \in F$. In other words, $solv(F, D)$ returns a new domain defined by

$$solv(F, D) = \text{gfp}(\lambda d.iter(F, d))(D) \qquad iter(F, D) = \sqcap_{f \in F} f(D).$$

where gfp denotes the greatest fixpoint w.r.t $\sqsubseteq$ lifted to functions.

3 SAT and unit propagation

A *proposition* p is a Boolean variable from a universe of Boolean variables $\mathcal{P}$. A *literal* l is either: a proposition p, its negation $\neg p$, the false literal $\bot$, or the true literal $\top$. The *complement* of a literal l, denoted $\neg l$, is $\neg p$ if $l = p$ or p if $l = \neg p$, while $\neg\bot = \top$ and $\neg\top = \bot$. A *clause* C is a disjunction of literals, which we also treat as a set of literals. An *assignment* is either a partial mapping μ from $\mathcal{P}$ to $\{\top, \bot\}$ or the failed assignment $\{\!\{\sqcup\}\!\}$. An assignment μ is treated as a set of literals $A = \{p \mid \mu(p) = \top\} \cup \{\neg p \mid \mu(p) = \bot\}$. We define a lattice over assignments as follows: $A \sqsubseteq A'$ iff $A \subseteq A'$ or $A' = \{\!\{\sqcup\}\!\}$. The least upper bound operation $\sqcup$ is defined as $A \sqcup A' = A \cup A$ unless either: the union contains $\{p, \neg p\}$ for some literal p, or one of A, A' is $\{\!\{\sqcup\}\!\}$, in which case $A \sqcup A' = \{\!\{\sqcup\}\!\}$.

An assignment A *satisfies* a clause C if one of the literals in C appears in A. A *theory* T is a set of clauses. An assignment is a *solution* to theory T if it satisfies each $C \in T$.

A SAT solver takes a theory T and determines if it has a solution. Complete SAT solvers typically involve some form of the Davis-Putnam-Logemann-Loveland algorithm [12] which combines search and propagation by recursively fixing the value of a proposition to either $\top$ (true) or $\bot$ (false) and using unit propagation to determine the logical consequences of each decision made so far. The unit propagation algorithm finds all unit resolutions of an assignment A with the theory T. Unit resolution of a clause $C = C' \cup \{l\}$ with assignment A, adds the literal l to A if the negation of each of the literals in C' occurs in A, since this is then the only way to satisfy C given the assignment A. Unit propagation continuously applies unit

resolution until the assignment A does not change. It can be formally defined as follows:

$$up(A, C) = \begin{cases} A \sqcup \{l\} & \exists l, C'.C = C' \cup \{l\}, \{\neg l' \mid l' \in C'\} \subseteq A \\ A & \text{otherwise} \end{cases}$$

$$UP(A, T) = \text{lfp}.(\lambda a. \bigsqcup_{C \in T} up(a, C))(A)$$

Example 4 Given the theory $T = \{ \neg p_1 \vee p_2 \vee p_3 \vee \neg p_4 \vee \neg p_5, p_1 \vee p_2, p_4 \vee \neg p_5\}$ and the assignment $A = \{\neg p_2, p_5\}$ unit propagation on $p_1 \vee p_2$ adds p_1, and on $p_4 \vee \neg p_5$ adds p_4, then unit propagation with the first clause adds p_3. Hence $UP(A, T) = \{p_1, \neg p_2, p_3, p_4, p_5\}$.

Modern DPLL SAT solvers combine very efficient unit propagation implemented using watched literal techniques together with 1UIP nogoods and backjumping [27] to create very powerful solvers.

4 Atomic constraints and propagation rules

Atomic constraints and propagation rules were originally devised for reasoning about propagation redundancy [6, 7]. They provide a way of describing the behaviour of propagators.

An *atomic constraint* represents the basic changes in domain that occur during propagation. For integer variables, the atomic constraints represent the elimination of values from an integer domain, i.e. $x_i \leqslant d$, $x_i \geqslant d$, $x_i \neq d$ or $x_i = d$ where $x_i \in \mathcal{V}_I$ and d is an integer. For set variables, the atomic constraints represent the addition of a value to a lower bound set of integers or the removal of a value from an upper bound set of integers, i.e. $e \in S_i$ or $e \notin S_i$ where e is an integer and $S_i \in \mathcal{V}_S$. We also consider the atomic constraint *false* which indicates that unsatisfiability is the direct consequence of propagation.

Define a *propagation rule* as $C \rightarrowtail c$ where C is a conjunction of *atomic constraints*, and c is a single atomic constraint such that $\not\models C \rightarrow c$. A propagation rule $C \rightarrowtail c$ defines a propagator (for which we use the same notation) in the obvious way.

$$(C \rightarrowtail c)(D)(v) = \begin{cases} \{\theta(v) \mid \theta \in D \cap soln(c)\} & \text{if } vars(c) = \{v\} \text{ and } \models D \rightarrow C \\ D(v) & \text{otherwise.} \end{cases}$$

In another words, $C \rightarrowtail c$ defines a propagator that removes values from D based on c only when D implies C. We can characterize an arbitrary propagator f in terms of the propagation rules that it implements. A propagator f *implements* a propagation rule $C \rightarrowtail c$ iff $\models D \rightarrow C$ implies $\models f(D) \rightarrow c$ for all $D \sqsubseteq D_{init}$.

Example 5 The propagator f_d for constraint $x \neq y$ of Example 2 implements the following propagation rules (among many others) for $D_{init}(x) = D_{init}(y) = \left[l \,..\, u \right]$.

$$x = d \rightarrowtail y \neq d,\ l \leqslant d \leqslant u$$
$$y = d \rightarrowtail x \neq d,\ l \leqslant d \leqslant u$$

Many propagators are better characterized by atomic constraints using bounds, e.g. $x \leqslant d$, although it is always possible to describe their behaviour using only the atomic constraints $x \neq d$.

Example 6 A common propagator f for the constraint $x_1 = x_2 \times x_3$ [24] is

$$\begin{aligned} f(D)(x_1) = {} & D(x_1) \cap \left[\min S \,..\, \max S \right] \\ & \text{where } S = \{(\min D(x_2)) \times (\min D(x_3)), (\min D(x_2)) \times (\max D(x_3)), \\ & (\max D(x_2)) \times (\min D(x_3)), (\max D(x_2)) \times (\max D(x_3))\} \\ f(D)(x_2) = {} & D(x_2) \text{ if } \min D(x_3) < 0 \wedge \max D(x_3) > 0 \\ & D(x_2) \cap \left[\min S \,..\, \max S \right] \text{ otherwise} \\ & \text{where } S = \{(\min D(x_1))/(\min D(x_3)), (\min D(x_1))/(\max D(x_3)), \\ & (\max D(x_1))/(\min D(x_3)), (\max D(x_1))/(\max D(x_3))\} \end{aligned}$$

and symmetrically for x_3.[2] Note that f does not enforce any notion of consistency.

The propagator f implements the following propagation rules (among many others) for $D_{init}(x_1) = D_{init}(x_2) = D_{init}(x_3) = [-20 \,..\, 20]$.

$$\begin{aligned} & x_2 \geqslant 2 \wedge x_3 \geqslant 3 \rightarrowtail x_1 \geqslant 6 \\ & x_1 \geqslant 6 \wedge x_3 \geqslant 0 \wedge x_3 \leqslant 3 \rightarrowtail x_2 \geqslant 2 \\ & x_1 \leqslant 10 \wedge x_2 \geqslant 6 \rightarrowtail x_3 \leqslant 1 \\ & x_1 \leqslant 10 \wedge x_2 \geqslant 9 \rightarrowtail x_3 \leqslant 1 \\ & x_2 \geqslant -1 \wedge x_2 \leqslant 1 \wedge x_3 \geqslant -1 \wedge x_3 \leqslant 1 \rightarrowtail x_1 \leqslant 1 \end{aligned}$$

Let $rules(f)$ be the set of all possible propagation rules implemented by f. A set of propagation rules $F \subseteq rules(f)$ *implements* f iff $solv(F, D) = f(D)$, for all $D \sqsubseteq D_{init}$.

This definition of $rules(f)$ is usually unreasonably large, and full of redundancy. For example the fourth propagation rule in Example 6 is clearly weaker than the third. In order to reason more effectively about propagation rules for a given propagator f, we seek a concise representation $rep(f) \subseteq rules(f)$ that implements f.

Example 7 The set of propagation rules given in Example 5 for the constraint $x \neq y$ define a minimal representation $rep(f_d)$ of the propagator f_d.

A propagation rule $C' \rightarrowtail c'$ is *directly redundant* with respect to another rule $C \rightarrowtail c$ if $D_{init} \models C' \rightarrow C \wedge c \rightarrow c'$ and not $D_{init} \models C \rightarrow C' \wedge c' \rightarrow c$. A propagation rule r for propagator f is *tight* if it is not directly redundant with respect to any rule in $rules(f)$. Obviously we would prefer to only use tight propagation rules in $rep(f)$ if possible.

[2] Division by zero has to be treated carefully here, see [24] for details.

Example 8 Consider the reified difference inequality $c \equiv x_0 \Leftrightarrow x_1 + 1 \leqslant x_2$ where $D_{init}(x_0) = \{0, 1\}$, $D_{init}(x_1) = \{0, 1, 2\}$, $D_{init}(x_2) = \{0, 1, 2\}$. Then a set of tight propagation rules $rep(f)$ implementing the domain propagator f for c is

$$\begin{array}{rr}
x_1 \leqslant 0 \wedge x_2 \geqslant 1 \rightarrowtail x_0 = 1 & x_1 \geqslant 2 \rightarrowtail x_0 = 0 \\
x_1 \leqslant 1 \wedge x_2 \geqslant 2 \rightarrowtail x_0 = 1 & x_1 \geqslant 1 \wedge x_2 \leqslant 1 \rightarrowtail x_0 = 0 \\
x_0 = 1 \rightarrowtail x_2 \geqslant 1 & x_2 \leqslant 0 \rightarrowtail x_0 = 0 \\
x_0 = 1 \wedge x_1 \geqslant 1 \rightarrowtail x_2 \geqslant 2 & x_0 = 0 \wedge x_1 \leqslant 1 \rightarrowtail x_2 \leqslant 1 \\
x_0 = 1 \rightarrowtail x_1 \leqslant 1 & x_0 = 0 \wedge x_1 \leqslant 0 \rightarrowtail x_2 \leqslant 0 \\
x_0 = 1 \wedge x_2 \leqslant 1 \rightarrowtail x_1 \leqslant 0 & x_0 = 0 \wedge x_2 \geqslant 1 \rightarrowtail x_1 \geqslant 1 \\
 & x_0 = 0 \wedge x_2 \geqslant 2 \rightarrowtail x_1 \geqslant 2
\end{array}$$

For constraints of the form $x_0 \Leftrightarrow x_1 + d \leqslant x_2$ we can build $rep(f)$ linear in the domain sizes of the variables involved.

A *bounds propagation rule* only makes use of atomic constraints of the form $x \leqslant d$, $x \geqslant d$ and *false*. We can classify a propagator f as a *bounds propagator* if it has a representation $rep(f)$ which only makes use of bounds propagation rules.

Example 9 The propagator in Example 8 is clearly a bounds propagator. A bounds propagator f_b for the constraint $x \neq y$ is defined by the propagation rules for $D_{init}(x) = D_{init}(y) = \left[l \,..\, u \right]$ where $l \leqslant d \leqslant u$:

$$\begin{array}{l}
x \leqslant d \wedge x \geqslant d \wedge y \leqslant d \rightarrowtail y \leqslant d - 1 \\
x \leqslant d \wedge x \geqslant d \wedge y \geqslant d \rightarrowtail y \geqslant d + 1 \\
y \leqslant d \wedge y \geqslant d \wedge x \leqslant d \rightarrowtail x \leqslant d - 1 \\
y \leqslant d \wedge y \geqslant d \wedge x \geqslant d \rightarrowtail x \geqslant d + 1.
\end{array}$$

5 Clausal representations of propagators

Propagators can be understood simply as a collection of propagation rules. This gives the key insight for understanding them as conjunctions of clauses, since we can translate propagation rules to clauses straightforwardly.

5.1 Atomic constraints and Boolean variables

Changes in domains of variables are the information recorded by a propagation solver. For example, $x = d$ is a change which fixes the value of x to domain value d, and $x \leqslant d$ is a change that restricts the value of x to be greater or equal to domain value d. In this sense atomic constraints are the "decisions" made or stored representing the sub-problem. In translating propagation to Boolean reasoning these decisions become the Boolean variables. For example, $[\![x = d]\!]$ and $[\![x \leqslant d]\!]$ denote Boolean variables which encode information about the possible values of a variable x with d an element in a (finite) integer domain.

We adopt an encoding of integer domains as Booleans which is concise and is well-matched to atomic constraints. It can be thought of as the combination of two well-studied representations: the *direct encoding* (see e.g. [35]) where the Boolean variables corresponding to integer variable x are of the form $[\![x = d]\!]$ for each value d in the domain and clauses are added to ensure that at most one such variable is true for a given x; and the *unary encoding* (see e.g. [2, 8]) where the Boolean variables are of the form $[\![x \leqslant d]\!]$ and clauses are added to ensure that $[\![x \leqslant d]\!]$ implies $[\![x \leqslant d']\!]$ for all $d' < d$ in the domain. A similar encoding is proposed by Ansótegui and Manyá in [1] where it is called the *regular encoding*. We encode set bounds domains as usual with Boolean variables of the form $[\![e \in S]\!]$ to represent that element e is in set S.

This choice of Boolean variables enables us to directly represent changes to domains made by atomic constraints. We define a mapping *lit* of atomic constraints to Boolean literals as follows:

$$
\begin{aligned}
&lit(false) &&= \bot \\
&lit(x_i = d) &&= [\![x_i = d]\!] && d \in D_{init}(x_i) \\
&lit(x_i \neq d) &&= \neg[\![x_i = d]\!] && d \in D_{init}(x_i) \\
&lit(x_i \leqslant d) &&= [\![x_i \leqslant d]\!] && \min D_{init}(x_i) \leqslant d < \max D_{init}(x_i) \\
&lit(x_i \leqslant d) &&= \top && d = \max D_{init}(x_i) \\
&lit(x_i \geqslant d) &&= \neg[\![x_i \leqslant d-1]\!] && \min D_{init}(x_i) < d \leqslant \max D_{init}(x_i) \\
&lit(x_i \geqslant d) &&= \top && d = \min D_{init}(x_i) \\
&lit(d \in S_i) &&= [\![d \in S_i]\!] && d \in \max D_{init}(S_i) \\
&lit(d \notin S_i) &&= \neg[\![d \in S_i]\!] && d \in \max D_{init}(S_i).
\end{aligned}
$$

where d is a value in the domain of variable x_i: $\min D_{init}(x_i) \leqslant d \leqslant \max D_{init}(x_i)$; and $e \in \max D_{init}(S_i)$. Note that *lit* is a bijection except where the result is $\top$, hence $lit^{-1}(l)$ is defined as long as $l \neq \top$.

There is a mapping from the domain of a variable v to an assignment on the Boolean variables $[\![x_i \leqslant d]\!]$, $[\![x_i = d]\!]$, and $[\![e \in S_i]\!]$ defined as:

$$
assign(D, v) = \begin{cases} \{\{\bot\}\} & D(v) = \varnothing \\ \{lit(c) \mid (v \in D(v)) \models c, v \in vars(c)\} & \text{otherwise} \end{cases}
$$

$$
assign(D) = \begin{cases} \{\{\bot\}\} & \exists v \in \mathcal{V}.D(v) = \varnothing \\ \bigcup_{v \in \mathcal{V}} assign(D, v) & \text{otherwise} \end{cases}
$$

Example 10 Consider the domain $D(x) = \{1, 3, 4\}$, $D(S) = [\,\{1, 2\} .. \{1, 2, 4\}\,]$ where $D_{init}(x) = [\,1 .. 6\,]$ and $D_{init}(S) = [\,\varnothing .. \{1, 2, 3, 4, 5\}\,]$. Since $x \in \{1, 3, 4\}$ implies $x \neq 2$, $x \geqslant 1$, $x \leqslant 4$, $x \leqslant 5$ and $x \leqslant 6$ then $assign(D, x) = \{\neg[\![x = 2]\!], [\![x \leqslant 4]\!], [\![x \leqslant 5]\!]\}$. Similarly since $S \in [\,\{1, 2\} .. \{1, 2, 4\}\,]$ implies $1 \in S$, $2 \in S$, $3 \notin S$ and $5 \notin S$ we have $assign(D, S) = \{[\![1 \in S]\!], [\![2 \in S]\!], \neg[\![3 \in S]\!], \neg[\![5 \in S]\!]\}$.

5.2 Faithfullness of domains

Given that we model constraint propagation in terms of Boolean variables of the form $[\![x_i \leqslant d]\!]$, $[\![x_i = d]\!]$, and $[\![e \in S_i]\!]$, it is necessary to insure that the Boolean

representation faithfully represents possible values of an integer variable. For example, the Boolean variables $[\![x = 3]\!]$ and $[\![x \leqslant 2]\!]$ cannot both take the value $\top$ (true).

To maintain faithfulness for an integer variable x where $D_{init}(x) = [\, l \,..\, u \,]$, we add two types of clauses: (a) for variables of the form $[\![x \leqslant d]\!]$ we add clauses to encode $[\![x \leqslant d]\!] \rightarrow [\![x \leqslant d+1]\!]$ (Eq. 1, below); and (b) for variables of the form $[\![x = d]\!]$ we add clauses to encode $[\![x = d]\!] \leftrightarrow ([\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d-1]\!])$ (Eqs. 2–6, below). In clause form, let $DOM(x)$ be the following clauses:

$$\neg[\![x \leqslant d]\!] \vee [\![x \leqslant d+1]\!] \quad l \leqslant d < u-1 \tag{1}$$

$$\neg[\![x = d]\!] \vee [\![x \leqslant d]\!] \quad l \leqslant d < u \tag{2}$$

$$\neg[\![x = d]\!] \vee \neg[\![x \leqslant d-1]\!] \quad l < d \leqslant u \tag{3}$$

$$[\![x = l]\!] \vee \neg[\![x \leqslant l]\!] \tag{4}$$

$$[\![x = d]\!] \vee \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d-1]\!] \quad l < d < u \tag{5}$$

$$[\![x = u]\!] \vee [\![x \leqslant u-1]\!] \tag{6}$$

The Boolean representation for set variables requires no additional clauses for faithfulness hence we define $DOM(S) = \{\}$ for a set variable S. Finally, we define the set of clauses (for all variables): $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

The faithfulness clauses $DOM(x)$ involve $2n$ Boolean variables and $4n$ clauses where n is the size of the domain $D_{init}(x)$. In contrast, the direct encoding of finite integer variables domains into SAT (which only involves the variables of the form $[\![x = d]\!]$) enforces faithfulness either with $O(n^2)$ clauses

$$(\vee_{d=l}^{u}[\![x = d]\!]) \bigwedge \wedge_{l \leqslant d_1 < d_2 \leqslant u}(\neg[\![x = d_1]\!] \vee \neg[\![x = d_2]\!]) \tag{7}$$

as described in [35] or takes the BDD approach described in [14] which requires a linear number of clauses but also introduces $2n$ fresh variables (in addition to the n original variables). Note that the regular encoding is linear, involves only $2n$ variables and has equally strong unit propagation as the quadratic encoding of the direct approach.

Theorem 1 *Let A be a set of literals on the variables $[\![x = d]\!], l \leqslant d \leqslant u$. Let T be the clauses of (7) Then $UP(DOM(x), A) = \{\{\bot\!\!\!\bot\}\}$ or $UP(T, A) \subseteq UP(DOM(x), A)$.*

Proof Let $A' = UP(DOM(x), A)$. Assume $A' \neq \{\{\bot\!\!\!\bot\}\}$ or we are done. The proof is by induction over the execution of $UP(T, A)$. Let $A = A_0 \subset A_1 \subset A_2 \subset \cdots \subset A_n = UP(T, A)$ be a sequence of assignments computed by $UP(T, A)$ where $A_{i+1} = A_i \cup \{l_i\} = up(A_i, C_i)$ for some l_i and $C_i \in T$. Assume that $A_i \subseteq A', i \leqslant m$. Consider $A_{m+1} = A_m \cup \{l_m\} = up(A_m, C_m)$. Suppose $l_m = \neg[\![x = d]\!]$ then C_m must be of the form $\neg[\![x = d]\!] \vee \neg[\![x = d']\!], d' \neq d$, so $[\![x = d']\!] \in A_m \subseteq A'$. Hence by (2) $[\![x \leqslant d']\!] \in A'$ and thus by (1) $[\![x \leqslant d'']\!] \in A', d'' \geqslant d'$, and by (3) $\neg[\![x = d'']\!] \in A', d'' > d'$. Similarly by (3) $\neg[\![x \leqslant d'-1]\!] \in A'$ and thus by (1) $\neg[\![x \leqslant d'']\!] \in A', d'' < d'$ and by (2) $\neg[\![x = d'']\!] \in A', d'' < d$. Clearly $\neg[\![x = d]\!] \in A'$ since either $d < d'$ or $d > d'$.

Suppose $l_m = [\![x = d]\!]$ then C_m must be the first clause in (7) and $\neg[\![x = d']\!] \in A_m \subseteq A', l \leqslant d' \neq d \leqslant u$. By (6) either $d = u$ or $[\![x \leqslant u-1]\!] \in A'$, and then by repeated use of (5) we have $[\![x \leqslant d']\!] \in A', d' \geqslant d$. By (4) either $d = l$ or $\neg[\![x \leqslant l]\!] \in A'$

and again by repeated use of (5) we have $\neg[\![x \leqslant d']\!] \in A', d' < d$. Finally using (5) (or (4) or (6) for the cases $d = l$ and $d = u$) we have $[\![x = d]\!] \in A'$.

A set of literals A can be converted to a domain:

$$domain(A)(v) = \begin{cases} \varnothing & A = \{\{\sqcup\}\} \\ \{d \in D_{init}(v) \mid & \forall [\![c]\!] \in A.vars(l) = \{v\} \Rightarrow v = d \models c \wedge \\ & \forall \neg[\![c]\!] \in A.vars(l) = \{v\} \Rightarrow v = d \models \neg c\} \\ & \text{otherwise} \end{cases}$$

that is the domain of all values for v that are consistent with all the Boolean variables related to v.

Example 11 Consider the set of literals $A = \{\neg[\![x = 2]\!], [\![x \leqslant 4]\!]\}$, where $D_{init}(x) = [1..6]$ then $x = 1 \models x \neq 2$ and $x = 1 \models x \leqslant 2$, similarly for $x = 3$ and $x = 4$, but not for $x = 2$, $x = 5$ and $x = 6$. Hence $domain(A)(x) = \{1, 3, 4\}$.

With domain clauses DOM, unit propagation on a translated set of atomic constraints generates all the consequences of the atomic constraints, i.e. faithfully represents a domain.

The following lemma shows that for an assignment A, that any atomic constraints that are a consequence of the decisions in A appear directly in the unit fixpoint of A with DOM.

Lemma 1 *Let $A' = UP(\varnothing, A \cup DOM(v))$. If $domain(A)(v) = S \neq \varnothing$ and $v \in S \models c$, then $lit(c) \in A'$. If $domain(A)(v) = \varnothing$, $A' = \{\{\sqcup\}\}$*

Proof We first consider a set variable $v = T$. If $domain(A)(T) = \varnothing$ then either $A = \{\{\sqcup\}\}$ or A must contain $[\![e \in T]\!]$ and $\neg[\![e \in T]\!]$ since $DOM(T) = \varnothing$.

Suppose $domain(A)(T) = \mathbf{S} \neq \varnothing$. Suppose $T \in \mathbf{S} \models e \in T$, then $e \in \bigcup_{S \in \mathbf{S}} S$. Clearly then $[\![e \in T]\!] \in A$ otherwise we can take any element of $d \in \mathbf{S}$ and then $d - \{e\} \in domain(A)(T)$ by definition. The result for $e \notin T$ is analogous.

Next we consider an integer variable $v = x$ The proof is by cases. Let $domain(A)(x) = S \neq \varnothing$.

Suppose $d \in D_{init}(x) - S$, then $x \in S \models x \neq d$. We show that $\neg[\![x = d]\!] \in A'$. Since $d \in D_{init}(x) - S$ then there is a literal in A which disallows the value. If it is (a) $\neg[\![x = d]\!] \in A$ we are done; if it is (b) $[\![x \leqslant d']\!] \in A, d' < d$ then by unit propagation on (1) we have $[\![x \leqslant d - 1]\!] \in A'$ and then propagating using (3) we have $\neg[\![x = d]\!] \in A'$; if it is (c) $\neg[\![x \leqslant d' - 1]\!] \in A, d' > d$ by unit propagation on (1) we have $\neg[\![x \leqslant d]\!] \in A'$ and then propagating using (2) we have $\neg[\![x = d]\!] \in A'$; and if it is $[\![x = d']\!] \in A, d' \neq d$ we have either $[\![x \leqslant d']\!] \in A', d' < d$ or $\neg x \leqslant d' - 1 \in A', d' > d$ using the (2) or (3) and then similar reasoning to case (b) and (c) applies.

Suppose $x \in S \models x \leqslant d$, we show that $[\![x \leqslant d]\!] \in A'$. Clearly we have $\neg[\![x = d']\!] \in A', \forall d' > d$ using the reasoning of the previous paragraph. Using unit propagation on the (6) we have $[\![x \leqslant u - 1]\!] \in A'$ and then using (5) we have $[\![x \leqslant d']\!] \in A', d' \geqslant d$. A similar argument applies if $x \in S \models x \geqslant d$ forcing $\neg[\![x \leqslant d' - 1]\!] \in A', d' \leqslant d$.

Finally suppose $S = \{d\}$ and hence $x \in S \models x = d$. Then using the previous paragraph we have $[\![x \leqslant d]\!] \in A'$ and $\neg[\![x \leqslant d - 1]\!] \in A'$ and using unit propagation on the (5) we have $[\![x = d]\!] \in A'$.

If $domain(A)(x) = \varnothing$, we show that unit propagation also leads to $\{\{\bot\!\!\!\bot\}\}$. Either $A' = \{\{\bot\!\!\!\bot\}\}$ and we are done or $A' \neq \{\{\bot\!\!\!\bot\}\}$. For each $d \in D_{init}(x)$ we have that $d \notin S$, so using the argument above $\neg[\![x = d]\!] \in A'$, $d \in D_{init}(x)$, and hence using arguments above $[\![x \leqslant d]\!] \in A', l \leqslant d < u$ Then using the (4) $[\![x = l]\!] \vee \neg[\![x \leqslant l]\!]$ we have a contradiction. Hence $A' = \{\{\bot\!\!\!\bot\}\}$. □

The following theorem shows that give a set of atomic constraints c on variable v, then the domain $D(v)$ that satisfies these constraints is isomorphic to the result of unit propagation on the faithfulness clauses DOM and the Boolean representation of the atomic constraints.

Theorem 2 *Let C be a set of atomic constraints on variable v, and $D(v) = \{d \mid v = d \models C\}$ then $assign(D, v) = UP(\varnothing, \{lit(c) \mid c \in C\} \cup DOM(v))$.*

Proof Let $A = UP(\varnothing, \{lit(c) \mid c \in C\} \cup DOM(v))$. If C is unsatisfiable, then $assign(D)(v) = \{\{\bot\!\!\!\bot\}\}$. Clearly $domain(\{lit(c) \mid c \in C\}, v) = \varnothing$, and hence by Lemma 1 we have $A = \{\{\bot\!\!\!\bot\}\}$.

Otherwise C is satisfiable, and $assign(D)(v) = \{lit(c') \mid vars(c') = \{x\}, C \models c'\}$.

We show $A \subseteq assign(D)(v)$ by induction on the unit propagation. Clearly the base case holds since the starting set is $\varnothing$. The first clauses $lit(c), c \in C$ can only add a literal $lit(c)$ which is in $assign(D)(v)$. The clauses in $DOM(v))$ can also add literals by unit propagation. Suppose the clauses (1) adds a literal, either $[\![x \leqslant d]\!] \in A$ and it adds $[\![x \leqslant d + 1]\!]$ or $\neg[\![x \leqslant d + 1]\!] \in A$ and it adds $\neg[\![x \leqslant d]\!]$. In the first case since $[\![x \leqslant d]\!] \in A \subseteq assign(D, v)$ we have that $C \models x \leqslant d$ and hence $C \models x \leqslant d + 1$ and the result holds. Similarly in the second case. $C \models \neg x \leqslant d + 1$ and hence $C \models \neg x \leqslant d$. Similar reasoning applies to unit propagations arising from the clauses representing $[\![x = d]\!] \leftrightarrow ([\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d - 1]\!])$.

We show $assign(D)(v) \subseteq A$. Clearly $domain(\{lit(c) \mid c \in C\}, v) = D(v)$. Hence c' where $vars(c') = \{v\}, C \models c'$ is such that $v \in D(v) \models c'$. By Lemma 1 we have that $lit(c') \in A$. □

Note that for the logarithmic encoding of an integer variable x as Booleans, where $D_{init}(x) = \left[0 \,..\, 2^k - 1\right]$ is encoded as $x = 2^{k-1}[\![x > 2^{k-1} - 1]\!] + 2^{k-2}[\![x \mod 2^{k-1} > 2^{k-2} - 1]\!] + \cdots + 2[\![x \mod 4 > 1]\!] + [\![x \mod 2 > 0]\!]$, a similar result to Theorem 2 is not possible since the encodings of atomic constraints are not single literals.

5.3 Propagation rules to clauses

The translation from propagation rules to clauses is straightforward:

$$cl(C \rightarrowtail c) = \vee_{c' \in C}(\neg\, lit(c')) \vee lit(c)$$

Example 12 The translation of the propagation rule:

$$x_2 \geqslant -1 \wedge x_2 \leqslant 1 \wedge x_3 \geqslant -1 \wedge x_3 \leqslant 1 \rightarrowtail x_1 \leqslant 1$$

is the clause $C_0 \equiv [\![x_2 \leqslant -2]\!] \vee \neg[\![x_2 \leqslant 1]\!] \vee [\![x_3 \leqslant -2]\!] \vee \neg[\![x_3 \leqslant 1]\!] \vee [\![x_1 \leqslant 1]\!]$. The advantage of the inequality literals is clear here: to define this clause using only $[\![x = d]\!]$ propositions for the domains given in Example 6 requires a clause of $\approx$100 literals.

The translation of propagation rules to clauses gives a system of clauses where unit propagation is at least as strong as the original propagators.

Theorem 3 *Let R be a set of propagation rules such that $D' = solv(R, D)$. Let $A = UP(assign(D), DOM \cup \bigcup\{cl(r) \mid r \in R\})$ then $A = \{\{\sqcup\}\}$ or $A \supseteq assign(D')$.*

Proof If $A = \{\{\sqcup\}\}$ we are done so assume A is a set of literals. Let

$$D = D_0, D_1 = r_1(D_0), D_2 = r_2(D_1), \ldots D_n = r_n(D_{n-1}) = solv(R, D)$$

be a sequence of propagations of individual rules $r_i \in R$ leading to the fixpoint D' We show by induction on i that $A \supseteq assign(D_i)$. The base case is obvious since $A \supseteq assign(D) = assign(D_0)$.

Suppose that $r_i \equiv c_1 \wedge \cdots \wedge c_m \rightarrowtail c$. If the rule did not fire then $D_i = D_{i-1}$ and we are done. Otherwise the rule fired and hence $D_{i-1} \models c_j, 1 \leqslant j \leqslant m$. Hence $lit(c_j) \in assign(D_{i-1})$. Now D_i is the domain D_{i-1} removing the values for v that do not satisfy c.

Now $\neg lit(c_1) \vee \cdots \vee \neg lit(c_m) \vee lit(c) \in cl(r_i)$ and hence unit propagation adds $lit(c)$ to A.

Clearly $A = UP(assign(D_{i-1}) \cup \{lit(c)\}, DOM \cup \bigcup\{cl(r) \mid r \in R\})$, because $assign(D) \subseteq assign(D_{i-1}) \cup \{lit(c)\} \subseteq A$. Now by Lemma 1 any c' where $v \in D_i(v) \models c'$ is such that $lit(c') \in A$. Hence $assign(D_i) \subseteq A$. □

In particular if we have clauses representing all the propagators F then unit propagation is guaranteed to be at least as strong as finite domain propagation.

Corollary 1 *Let $rep(f)$ be a set of propagation rules implementing propagator f. Let $A = UP(assign(D), DOM \cup \bigcup\{cl(r) \mid f \in F, r \in rep(f)\})$. Then $A = \{\{\sqcup\}\}$ or $A \supseteq assign(solv(F, D))$.*

Example 13 Notice that the clausal representation may be "stronger" than the propagator. Consider the propagator f for $x_1 = x_2 \times x_3$ defined in Example 6. Then the clause C_0 defined in Example 12 is in the Boolean representation of the propagator. Given $\neg[\![x_2 \leqslant -2]\!], [\![x_2 \leqslant 1]\!], \neg[\![x_3 \leqslant -2]\!], \neg[\![x_1 \leqslant 1]\!]$ we infer $\neg[\![x_3 \leqslant 1]\!]$. But given the domain $D(x_1) = [\,2\,..\,20\,]$, $D(x_2) = [\,-1\,..\,1\,]$, and $D(x_3) = [\,-1\,..\,20\,]$ then $f(D)(x_3) \neq [\,2\,..\,20\,]$. In fact the propagator f can determine no new information.

Given the Corollary above it is not difficult to see that, if it uses the same search strategy as a propagation based solver for propagators F, a SAT solver using clauses $\bigcup\{cl(r) \mid f \in F, r \in rep(f)\})$ needs no more search space to find the same solution(s).

But there is a difficulty in this approach. Typically $rep(f)$ is extremely large. The size of $rep(f)$ for the propagator f for $x_1 = x_2 \times x_3$ of Example 6 is around 100,000 clauses. But clearly most of the clauses in $rep(f)$ must be useless in any computation, otherwise the propagation solver would make an enormous number of propagation

steps, and this is almost always not the case. This motivates the fundamental idea of this paper which is to represent propagators lazily as clauses, only adding a clause to its representation when it is able to propagate new information.

6 Lazy clause generation

The key idea is, rather than *a priori* representing a propagator f by a set of clauses, to execute the propagator during the SAT search and record which propagation rules actually fired as clauses.

We execute a SAT solver over theory $T \supseteq DOM$. At each fixpoint of unit propagation we have an assignment A which corresponds to a domain $D = domain(A)$. We then execute (individually) each propagator $f \in F$ on this domain obtaining a new domain $D' = f(D)$. We then select a set of propagation rules R implemented by f such that $solv(R, D) = D'$ and add the clauses $\{cl(r) \mid r \in R\}$ to the theory T in the SAT solver. In fact we add these clauses to the SAT solver one by one because adding a single new clause may cause failure which means the rest of the work is avoided.

Given the above discussion we modify our propagators, so that rather than returning a new domain they return a set of propagation rules that would fire adding new information to the domain.

Let $lazy(f)$ be the function from domains to sets of propagation rules $R \subseteq rules(f)$ such that if $f(D) = D'$ then $lazy(f)(D) = R$ where $solv(R, D) = D'$, and for each $C \rightarrowtail c \in R$ it is not the case that $D \models c$ (that is each rule in R generates new information).

With the lazy version of a propagator defined we can define lazy propagation as Algorithm 1. We repeatedly search for a propagator which is not at fixpoint and add the clausal version of a propagation rule that will fire using the lazy version of the propagator.

Algorithm 1 LAZY_PROP(A,F,T)

Input: A is an assignment, F is a set of propagators, T is set of clauses including DOM
Output: (A', T') an assignment $A' \supseteq A$ or $\{\{\bot\}\}$ and a set of clauses $T' \supseteq T$

```
1  repeat
2    A := UP(A, T) ;
3    T_0 := T ;
4    D := domain(A) ;
5    for all f ∈ F do
6      if f(D) ≠ D then
7        let r ∈ lazy(f)(D);
8        T := T ∪ {cl(r)} ;
9        break
10 until T = T_0 ;
11 return (A; T)
```

We are interested in minimal assignments that model a domain D to automatically create lazy versions of propagators. Let $A = UP(A, DOM(v))$, then an

information equivalent assignment is any A' where $A = UP(A', DOM(v))$. Define $minassign(A, v)$ as the set A' of minimal cardinality where $A = UP(A', DOM(v))$, and preferring positive equational literals, over inequality literals, over negative equational literals.

Example 14 The set $A = \{[\![x = 1]\!], [\![x \leqslant 1]\!], \neg[\![x \geqslant 2]\!], \neg[\![x = 0]\!], \neg[\![x = 2]\!]\}$ is a fixpoint of $DOM(x)$ assuming $D_{init}(x) = [0..2]$. $minassign(A, x) = \{[\![x = 1]\!]\}$, since $A = UP(\{[\![x = 1]\!]\}, DOM(x))$.

The set $A' = \{[\![x \leqslant 1]\!], \neg[\![x = 2]\!]\}$ is also a fixpoint of $DOM(x)$. Here $minassign(A, x) = \{[\![x \leqslant 1]\!]\}$ even though $A' = UP(\{\neg[\![x = 2]\!]\}$ is information equivalent, because inequalities are preferred over negated equality literals.

We can automatically create $lazy(f)$ from f as follows. Let $f(D) = D'$ and let $C_v = minassign(D', v) - assign(D, v)$ be the new information (propositions) about v determined by propagating f on domain D. Then a correct set of rules $R = lazy(f)(D)$ is the set of propagation rules

$$\wedge_{v \in input(f)}\{lit^{-1}(l') \mid l' \in minassign(D, v)\} \rightarrowtail lit^{-1}(l)$$

for each $v \in output(f)$ and each $l \in C_v$

We can almost certainly do better than this. Usually a propagator is well aware of the reasons why it discovered some new information. The following examples illustrates how we can improve upon the default lazy version of a propagator.

Example 15 Consider the propagator f for $x_1 = x_2 \times x_3$ defined in Example 6. Applied to $D(x_1) = [-10..18]$, $D(x_2) = \{3, 5, 6\}$, $D(x_3) = [1..3]$ it determines $f(D)(x_1) = [3..18]$. The new information is $\neg[\![x_1 \leqslant 2]\!]$. The naive propagation rule defined above is

$$x_1 \geqslant -10 \wedge x_1 \leqslant 18 \wedge x_2 \geqslant 3 \wedge x_2 \neq 4 \wedge x_2 \leqslant 6 \wedge x_3 \geqslant 1 \wedge x_3 \leqslant 3 \rightarrowtail x_1 \geqslant 3$$

It is easy to see from the definition of the propagator, that the bounds of x_1 and the missing values in x_2 are irrelevant, so the propagation rule could be

$$x_2 \geqslant 3 \wedge x_2 \leqslant 6 \wedge x_3 \geqslant 1 \wedge x_3 \leqslant 3 \rightarrowtail x_1 \geqslant 3$$

but in fact it could also correctly simply be $x_2 \geqslant 3 \wedge x_3 \geqslant 1 \rightarrowtail x_1 \geqslant 3$ but this is not so obvious from the definition of f. The final rule is tight.

Example 16 Consider the propagator f for $x_0 \leftrightarrow x_1 + 1 \leqslant x_2$ from Example 8. When applied to the domain $D(x_0) = \{0, 1\}$, $D(x_1) = \{1, 2\}$, $D(x_2) = \{0\}$ it determines $f(D)(x_0) = \{0\}$. We can define $lazy(f)$ to return propagation rules in $rep(f)$ as defined in Example 8. For this case $lazy(f)(D)$ could return either $\{x_1 \geqslant 1 \wedge x_2 \leqslant 1 \rightarrowtail x_0 = 0\}$ or $\{x_2 \leqslant 0 \rightarrowtail x_0 = 0\}$.

Given we understand the implementation of propagator f, it is usually straightforward to see how to implement $lazy(f)$.

Example 17 Let $c \equiv \sum_{i=1}^{n} a_i x_i - \sum_{i=n+1}^{m} b_i x_i \leqslant d$ be a linear constraint where $a_i > 0$, $b_i > 0$. The bounds propagator f for c is defined as

$$f(D)(x_i) = D(x_i) \cap \left[-\infty \,..\, \lfloor \tfrac{S - a_i \min D(x_i)}{a_i} \rfloor \right] \quad 1 \leqslant i \leqslant n$$
$$f(D)(x_i) = D(x_i) \cap \left[\lceil \tfrac{S - b_i \max D(x_i)}{b_i} \rceil \,..\, +\infty \right] \quad n+1 \leqslant i \leqslant m$$

where $S = d - \sum_{i=1}^{n} a_i \min D(x_i) + \sum_{i=n+1}^{m} b_i \max D(x_i)$. If the bounds changes for some $x_i, 1 \leqslant i \leqslant n$, so $u_i = \max f(D)(x_i) < \max D(x_i)$ then the propagation rule $lazy(f)$ generates is

$$\bigwedge_{j=1,\, j\neq i}^{n} x_i \geqslant \min D(x_i) \wedge \bigwedge_{j=n+1}^{m} x_i \leqslant \max D(x_i) \rightarrowtail x_i \leqslant u_i$$

similarly for $x_i, n+1 \leqslant i \leqslant m$. Note that this is not necessarily tight.

We claim extending a propagator f to create $lazy(f)$ is usually straightforward. For example, Katsirelos and Bacchus [20] explain how to create $lazy(f)$ (or the equivalent in their terms) for the `alldifferent` domain propagator f by understanding the algorithm for f. For a propagator f defined by indexicals [34], we can straightforwardly construct $lazy(f)$ since the indexical definition illustrates directly which atomic constraints contributed to the result. Direct constructions of $lazy(f)$ may not necessarily be tight. For propagators implemented using Binary Decision Diagrams we can automatically generate tight propagation rules using BDD operations [17]. If we want to generate tight propagation rules from arbitrary propagators f then we may need to modify the algorithm for f more substantially to obtain $lazy(f)$.

Example 18 We can make the propagation rules of Example 17 tight by weakening the bounds on some other variables. Let $r = a_i(u_i + 1) - (S - a_i \min D(x_i)) - 1$ be the remainder before rounding down will increase the bound. If there exists $a_j \leqslant r$ where $\min D(x_j) > \min D_{init}(x_j)$ then we can weaken the propagation rule replacing the atomic constraint $x_j \geqslant \min D(x_j)$ by $x_j \geqslant \min D(x_j) - r_j$ where $r_j = \min\{\lfloor \frac{r}{a_j} \rfloor, \min D(x_j) - \min D_{init}(x_j)\}$. This reduces the remainder r by $a_j r_j$. Similarly if there exists $b_j \leqslant r$. We can repeat the process until $r < a_j$ and $r < b_j$ for all j. The result is tight.

For example given $100x_1 + 50x_2 + 10x_3 + 9x_4 \leqslant 100$ where $D_{init}(x_1) = D_{init}(x_2) = D_{init}(x_3) = D_{init}(x_4) = [-3 \,..\, 10]$ where $D(x_1) = D(x_2) = D(x_3) = D(x_4) = [0 \,..\, 10]$ then the propagation gives $S = 100$. The new upper bound on x_1 is $u_1 = 1$, and $r = 100 \times 2 - (100 - 100 \times 0) - 1 = 99$. The initial propagation rule is

$$x_2 \geqslant 0 \wedge x_3 \geqslant 0 \wedge x_4 \geqslant 0 \rightarrowtail x_1 \leqslant 1$$

We have $a_2 < r$ so we can decrease the coefficient of x_2 by $\min\{\lfloor \frac{99}{50} \rfloor, 3\} = 1$. There is still a remainder of $r = 99 - 1 \times 50 = 49$. We can reduce the coefficient of x_3 by 3 (the maximum since this takes it to the initial lower bound). This still leaves $r = 49 - 3 \times 10 = 19$. We can reduce the coefficient of x_4 by 2, the remainder is now 1, and less than any coefficient. The final tight propagation rule is

$$x_2 \geqslant -1 \wedge x_3 \geqslant -3 \wedge x_4 \geqslant -2 \rightarrowtail x_1 \leqslant 1$$

Regardless of the tightness of propagation rules, the lazy clause generation approach ensures that the unit propagation that results is at least as strong as applying the propagators themselves.

Theorem 4 *Let* $(A, T) = \text{LAZY\_PROP}(assign(D), F, DOM)$ *then* $A = \{\{\bot\}\}$ *or* $A \supseteq assign(solv(F, D))$.

Proof If $A = \{\{\bot\}\}$ we are done, so assume $A \neq \{\{\bot\}\}$. By definition of LAZY_PROP, if $D' = domain(A)$ then $f(D') = D'$ otherwise LAZY_PROP would have added a new propagation rule to the theory T. Hence $D' = solv(F, D')$ and clearly $D' \sqsubseteq solv(F, D)$. since $solv(F, D)$ is the largest mutual fixpoint of $f \in F$ less than D. Thus $A = assign(D') \supseteq assign(solv(F, D))$. □

Because we only execute the propagators at a fixpoint of unit propagation, generating a propagation rule whose right hand side gives new information means the clause cannot previously occur. The advantage of tight propagators is that, if the set of propagation rules R generated by $lazy(f)$ is tight, over the lifetime of a search it will not involve any direct redundancy.

7 Choices for modelling in lazy clause generation

Lazy clause generation combines a SAT solver with a finite domain solver. Because we have two solvers available a whole range of possibilities arise in modelling a constraint problem. In this section we explore some of the modelling possibilities that the novel solving technology of lazy clause generation allows.

7.1 Laziness and eagerness

An important choice in the lazy clause generation approach is whether to implement a propagator lazily (which is the default) or eagerly. The eager representation of a propagator f simply adds the clauses $cl(r)$ for all $r \in rep(f)$ into the SAT solver before beginning the search. This clearly can improve search, since more information is known *a priori*, but the size of the clausal representation may make it inefficient.

Example 19 The representation of the domain propagator for disequality $x \neq y$ where $D_{init}(x) = D_{init}(y) = [l\,..\,u]$ requires $2(u - l + 1)$ binary clauses. Hence it is possible to model eagerly.

The representation of the bounds propagator for $x_1 + \cdots + x_n \leqslant k$ where $D_{init}(x_1) = \cdots = D_{init}(x_n) = [0\,..\,1]$ has

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

propagation rules. Clearly it is impossible to represent this eagerly for large n and k.

In practice eager representation is only useful for constraints that have small representations.

7.2 Variable representation

The lazy clause generation approach represents variables domains of possible values in dual manner: a Boolean assignment and a domain D on integer variables. There are a number of choices of how we can represent integer variables in terms of Boolean variables. The default choice (full integer representation) is described in the Section 5. We present new choices below.

7.2.1 Non-continuous variables

We can represent an integer variable where $D_{init}(x) = \{d_1, \ldots, d_n\}$ where $d_i < d_{i+1}, 1 \leqslant i \leqslant n$, and the values are noncontinuous. This requires fewer Boolean variables, and fewer domain constraints than representing the domain $\left[d_1 \,..\, d_n \right]$. The Boolean representation uses variables $[\![x = d_i]\!], 1 \leqslant i \leqslant n$ and $[\![x \leqslant d_i]\!], 1 \leqslant i < n$.

The clauses $DOM(x)$ required to maintain faithfulness of the Boolean assignment are:

$$\neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i+1}]\!] \quad 1 \leqslant i < n-1 \tag{8}$$

$$\neg[\![x = d_i]\!] \vee [\![x \leqslant d_i]\!] \quad 1 \leqslant i < n \tag{9}$$

$$\neg[\![x = d_i]\!] \vee \neg[\![x \leqslant d_{i-1}]\!] \quad 1 < i \leqslant n \tag{10}$$

$$[\![x = d_1]\!] \vee \neg[\![x \leqslant d_1]\!] \tag{11}$$

$$[\![x = d_i]\!] \vee \neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i-1}]\!] \quad 1 < i < n \tag{12}$$

$$[\![x = d_n]\!] \vee [\![x \leqslant d_{n-1}]\!] \tag{13}$$

7.2.2 Bounds variables

We can represent an integer variable only using the bounds variables $[\![x \leqslant d]\!], l \leqslant d < u$ where $D_{init}(x) = \left[l \,..\, u \right]$. While this means we cannot represent all possible subsets of $\left[l \,..\, u \right]$, it has the advantage of requiring fewer Boolean variables, and the domain representation requires only the clauses (1):

$$\neg[\![x \leqslant d]\!] \vee [\![x \leqslant d+1]\!] \quad l \leqslant d < u-1$$

7.2.3 Non-continuous bounds variables

We can clearly restrict the representation of non-continuous variables to bounds only analogously, just using the Boolean variables $[\![x \leqslant d_i]\!]$ and the clauses (8).

7.3 Propagator and variable representation independence

In a usual finite domain solver we are restricted so that if we use bounds variables, they must be restricted to only occur in bounds propagators. Indeed we can use this observation to avoid using full integer variables for variables that only occur in bounds propagators. In the lazy clause generation solver we can separate the Boolean variable representation from the propagator type. This is because the propagator works on the domains representing the variables, and this is distinct from the Boolean representation of domains.

With this separation the propagation engine can work without knowing whether integer variable x is a full integer, non-continuous, or bounds variable, since the translation of assignments to domains, and from propagation rules to clauses, completely captures the relationship between the Boolean representation and the integer variable.

Because of this separation we can independently choose which propagator we will use to represent a problem, without considering the Boolean variable representation. Hence for an individual constraint we can choose any of the propagators for that constraint.

7.3.1 Non-continuous variables

We extend the translation of atomic constraints *lit* to map atomic constraints involving non-continuous variable x where $D_{init}(x) = \{d_1, \ldots, d_n\}$ as follows:

$$lit(x = d) = \begin{cases} \bot & d \notin \{d_1, \ldots, d_n\} \\ [\![x = d_i]\!] & d = d_i \end{cases}$$

$$lit(x \neq d) = \begin{cases} \top & d \notin \{d_1, \ldots, d_n\} \\ \neg[\![x = d_i]\!] & d = d_i \end{cases}$$

$$lit(x \leqslant d) = \begin{cases} \top & d >= d_n \\ \bot & d < d_1 \\ [\![x \leqslant d_i]\!] & d_i < d \leqslant d_{i+1} \end{cases}$$

$$lit(x \geqslant d) = \begin{cases} \top & d \leqslant d_1 \\ \bot & d > d_n \\ \neg[\![x \leqslant d_i]\!] & d_i < d \leqslant d_{i+1} \end{cases}$$

Note that each atomic constraint is translated as a single literal.

Example 20 Consider the translation of the propagation rules $x = 3 \rightarrowtail y \neq 3$ and $x \neq 3 \rightarrowtail y = 3$, where $D_{init}(x) = \{0, 3, 5\}$ and $D_{init}(y) = \{1, 2, 4\}$. The resulting clauses are $\neg[\![x = 3]\!] \vee \top$ or $\top$ (the always true clause) and $[\![x = 3]\!] \vee \bot$ or equivalently $[\![x = 3]\!]$.

7.3.2 Bounds variables

We extend the translation of atomic constraints *lit* to map atomic constraints involving bounds variable x where $D_{init}(x) = \left[\, l \,..\, u \,\right]$ as follows:

$$lit(x = d) = \begin{cases} [\![x \leqslant d]\!] & d = l \\ [\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d - 1]\!], & l < d < u \\ \neg[\![x \leqslant u - 1]\!] & d = u \\ \bot & \text{otherwise} \end{cases}$$

$$lit(x \neq d) = \begin{cases} \neg[\![x \leqslant d]\!] & d = l \\ \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d - 1]\!], & l < d < u \\ [\![x \leqslant u - 1]\!] & d = u \\ \top & \text{otherwise} \end{cases}$$

The translations of $x \leqslant d$ and $x \geqslant d$ are as for full integer variables. Note that these translations now no longer guarantee to return a single literal.

Clearly "Boolean integer" variables x where $D_{init}(x) = [\,0\,..\,1\,]$ can be represented as bounds only variables without loss of expressiveness since $x \leqslant 0 \leftrightarrow x = 0 \leftrightarrow \neg(x = 1)$.

We can translate any propagation rule to a conjunction of clauses by simply applying *lit* as before. This creates (a possibly non-clausal) Boolean formulae which can be transformed to conjunctive normal form.

Example 21 Consider the translation of the propagation rule $x = 3 \rightarrowtail y \neq 3$, where x and y are bounds only variables. The resulting formula is $\neg[\![x \leqslant 3]\!] \vee [\![x \leqslant 2]\!] \vee [\![y \leqslant 2]\!] \vee \neg[\![y \leqslant 3]\!]$, which is a clause already.

Consider the translation of the propagation rule $x \neq 3 \rightarrowtail y = 3$. The resulting formula is $\neg([\![x \leqslant 2]\!] \vee \neg[\![x \leqslant 3]\!]) \vee ([\![y \leqslant 3]\!] \wedge \neg[\![y \leqslant 2]\!])$. The conjunctive normal form is

$$\begin{aligned} \neg[\![x \leqslant 2]\!] \vee [\![y \leqslant 3]\!] \\ [\![x \leqslant 3]\!] \vee [\![y \leqslant 3]\!] \\ \neg[\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 2]\!] \\ [\![x \leqslant 3]\!] \vee \neg[\![y \leqslant 2]\!] \end{aligned}$$

It would appear that the conversion of propagation rules including bounds variables could lead to an exponential explosion in the number of clauses required to represent them. By restricting the conversion of the rules to clauses which may actually be able to cause unit propagation, in fact we can represent them with at most 2 clauses.

Lemma 2 *If domain $D = domain(A)$ is such that $D(x) \models x \neq d$ where x is a bounds only variable, then $D(x) \models x \geqslant d + 1$ or $D(x) \models x \leqslant d - 1$.*

Proof Now A can only include literals $[\![x \leqslant d']\!]$ or $\neg[\![x \leqslant d']\!]$ for some d'. Hence $domain(A)(x)$ is a range domain. If $D(x) \models x \neq d$ then either $D(x) \models x \geqslant d + 1$ or $D(x) \models x \leqslant d - 1$. □

Define the bounds simplification $bs(r)$ of a propagation rule $r \equiv C \rightarrowtail c$, for domain $D = domain(A)$ for some assignment A which fires the rule, as follows. Replace each atomic constraint $x \neq d$ appearing in C where x is a bounds only variable by either $x \leqslant d - 1$ or $x \geqslant d + 1$, whichever holds in D. The resulting propagation rule can create at most 2 clauses.

Theorem 5 *The conjunctive normal form of the clausal representation of $bs(r)$ involves at most 2 clauses.*

Proof Each atomic constraint appearing in the left hand side of $bs(r)$ is translated as a single Boolean literal. The only conjunction that can occur in the translation is if the right hand side is an atomic constraint $x = d$ and x is a bounds variable. The resulting CNF has two clauses. □

Example 22 Consider the translation of the propagation rule $r \equiv x \neq 3 \rightarrowtail y = 3$ where x and y are bounds variables ranging over $[0..10]$. Suppose the domain that causes it to fire is $D = domain(A)$ where $A = \{[\![x \leqslant 1]\!]\}$. Then $D(x) = [0..1]$ and $D(x) \models x \leqslant 2$ and $bs(r) \equiv x \leqslant 2 \rightarrowtail y = 3$. The translation to Booleans is the formula $\neg[\![x \leqslant 2]\!] \vee ([\![y \leqslant 3]\!] \wedge \neg[\![y \leqslant 2]\!])$, which in CNF is $(\neg[\![x \leqslant 2]\!] \vee [\![y \leqslant 3]\!]) \wedge (\neg[\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 2]\!])$. Note that the two clauses from Example 21 that are missing could not fire in A.

There is an important new behaviour that arises when we consider using domain propagators on bounds variables. The result of propagation is always a clause of a form

$$cl(C \rightarrowtail c) = \vee_{c' \in C}(\neg\, lit(c')) \vee lit(c),$$

where $\neg\, lit(c')$ are all false in the current assignment and $lit(c)$ is either undefined or false in the current assignment. Previously $lit(c)$ was always a single literal, hence we could guarantee unit propagation would apply, and set $lit(c)$ to true. Now there is a possibility that $lit(c)$ is itself a disjunction and unit propagation will not apply.

Example 23 Consider the execution of the domain propagation for $x \neq y$ (Example 2) where x and y are bounds variables on the assignment $A = \{[\![x \leqslant 3]\!], \neg[\![x \leqslant 2]\!]\}$. Then in the corresponding $domain(A)(x) = \{3\}$ and the propagation rule $x = 3 \rightarrowtail y \neq 3$ fires. The resulting clause is $\neg[\![x \leqslant 3]\!] \vee [\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 3]\!] \vee [\![y \leqslant 2]\!]$. No unit propagation is possible using A and this new clause.

In fact the domain propagator for $x \neq y$ applied to bounds variables x and y generates exactly the same clauses as the bounds propagator, but it generates them earlier!

7.4 Disjunctive propagators

The discussion of the end of the last subsection motivates examining a new possibility. Propagation rules are designed so that the result of the propagation is a single atomic constraint, which can then be represented immediately as a change in domain. Given that we will convert the propagation rules to clauses in any case we can extend them to allow disjunction on the right hand side. A *disjunctive propagation rule* has the form $c_1 \wedge \cdots \wedge c_n \rightarrowtail c_{n+1} \vee \cdots \vee c_{n+m}$. The translation to clauses is clear $cl(c_1 \wedge \cdots \wedge c_n \rightarrowtail c_{n+1} \vee \cdots \vee c_{n+m}) = \neg\, lit(c_1) \vee \cdots \vee \neg\, lit(c_n) \vee lit(c_{n+1}) \vee \cdots \vee lit(c_{n+m})$. Presently we restrict our implementation to only support disjunctive propagation rules with at most two literals on the right hand side.

Example 24 Consider the constraint $|x - y| \geqslant k$ for constant $k > 0$. The bounds propagator for this constraint has representation given by the propagation rules: (where $l + k > u - k$)

$$x \geqslant l \wedge x \leqslant u \wedge y \leqslant l + k - 1 \rightarrowtail y \leqslant u - k$$
$$x \geqslant l \wedge x \leqslant u \wedge y \geqslant u - k + 1 \rightarrowtail y \geqslant l + k$$
$$y \geqslant l \wedge y \leqslant u \wedge x \leqslant l + k - 1 \rightarrowtail x \leqslant u - k$$
$$y \geqslant l \wedge y \leqslant u \wedge x \geqslant u - k + 1 \rightarrowtail x \geqslant l + k$$

A more eager version of this propagator fires when the range on one variable is small enough to guarantee some (disjunctive) constraints on the other variable. It is defined by the disjunctive propagation rules: (where $l + k > u - k$)

$$x \geqslant l \wedge x \leqslant u \rightarrowtail y \geqslant l + k \vee y \leqslant u - k$$
$$y \geqslant l \wedge y \leqslant u \rightarrowtail x \geqslant l + k \vee x \leqslant u - k$$

Disjunctive propagators can be seen as a more eager form of lazy clause generation.

We shall see in the next section that disjunctive propagators (including those resulting from domain propagation on bounds variables) do complicate things considerably, principally because we are not guaranteed that they will cause unit propagation when they are added to the SAT solver.

8 Building a lazy clause generator system

The creation of a practical lazy clause generation solver involves many more considerations than were addressed in Section 6. To build the system we add a cut down propagation engine into a SAT solver and modify it as a lazy clause generator.

The SAT solver applies unit propagation, and when it reaches a fixpoint it calls the propagation engine. The new literals set by the SAT solver are converted into domain changes in the propagation solver, and these "events" queue up propagators for execution. Each Boolean literal in the SAT solver actually corresponds to a different event in a propagation engine: $[\![x \leqslant d]\!]$ and $[\![e \in T]\!]$ corresponds to lower bound change events, $\neg[\![x \leqslant d]\!]$ and $\neg[\![e \in T]\!]$ correspond to upper bound change events, $[\![x = d]\!]$ corresponds to a variable fixing event, and $\neg[\![x = d]\!]$ corresponds to a domain change event.

The first propagator in the queue is then executed. If it causes propagation, then the clausal representation of the first propagation rule that fires is added to the SAT solver and unit propagation is applied. When the SAT solver finishes we re-execute the same propagator (which is still at the head of the queue) to search for another firing propagation rule. When there are no more firing rules the propagator is removed from the queue and the next propagator considered. The reason we add clauses as soon as possible is to detect failure as soon as possible. Unit propagations may schedule (or re-schedule) propagators. The process continues until the propagation queue is empty and unit propagation is at fixpoint. At this point the SAT solver makes a decision about a literal to set *true* and search continues.

On failure the propagation queue is cleared, and the SAT solver backtracks up the trail of decided and inferred literals. For each canceled Boolean literal l which is removed from the current assignment, we undo the change of atomic constraint $lit^{-1}(l)$ to the domain D. Note that since all individual domain changes are reflected in Boolean literals this is sufficient.

Example 25 Suppose $[\![x \leqslant 5]\!]$ was inferred at an earlier point in execution so $\max D(x) = 5$. Then suppose $[\![x \leqslant 2]\!]$ is inferred. In forward execution we will

modify max $D(x) = 2$, but unit propagation will also infer $[\![x \leqslant 3]\!]$ and $[\![x \leqslant 4]\!]$. On backtracking we walk up the trail of decided and inferred variables. When we unset $[\![x \leqslant 4]\!]$ we reset max $D(x) = 5$, and then when unsetting $[\![x \leqslant 3]\!]$ and $[\![x \leqslant 2]\!]$ we do not change it further.

A subtle point we have not addressed is why we do not worry about a propagator creating duplicates of clauses corresponding to its propagation rules, particularly since we can execute the propagator repeatedly simply to create all the propagation rules that fire for one domain. The reason is that since a propagator f is only run at domain $D = domain(A)$ for an assignment A which is a unit propagation fixpoint, then if $cl(r)$ is already in the SAT solver then r cannot fire on domain D (it has no new information).

Example 26 Consider the propagation of the constraint $x = y$ with $D_{init}(x) = D_{init}(y) = [0..4]$. After the SAT solver sets $\neg[\![x = 2]\!]$ and $\neg[\![y = 3]\!]$ the first propagation rule that fires is $x \neq 2 \rightarrowtail y \neq 2$. This is added as the clause $[\![x = 2]\!] \vee \neg[\![y = 2]\!]$ and propagated to set $\neg[\![y = 2]\!]$. Returning to the propagation engine, the propagator for $x = y$ is still at the head of the queue. The original propagation rule no longer fires since $y \neq 2$ is not new information. Hence the next propagation rule $y \neq 3 \rightarrowtail x \neq 3$ is considered.

When we extend lazy clause generation to allow domain propagators on bounds variables, or more generally disjunctive propagators the considerations above fail to hold. The reason is that the resulting newly added clauses may not cause unit propagation with the current assignment. Hence we can add the clauses multiple times.

This requires two modifications to the approach. First disjunctive propagators at the head of the queue must store an index of the propagation rule processed last, and clear this index every time the propagator queue is cleared. This is to avoid them regenerating the same propagation rule when they are still the head of the queue. Secondly, before adding a clause corresponding to a disjunctive propagation rule we need to check that it is not already in the SAT solver.

We could build a separate data structure to record which clauses have been sent to the solver. To avoid the complexity and space required to do this we re-use existing data structures in the SAT solver. The following approach relies on the restriction that the right hand side of a disjunctive propagation rule has at most two literals. This is clearly the case for all disjunctive propagators resulting from domain propagation on bounds variables.

Suppose a disjunctive propagation rule $C \rightarrowtail c_1 \vee c_2$ already has its corresponding clause Cl in the SAT solver. All literals in the clause except $lit(c_1)$ and $lit(c_2)$ must be false in the current assignment, otherwise the propagation rule would not fire. The SAT solver keeps track of at least two literals in each clause which are not false, the so-called *watched literals*, in order to detect unit propagations. Hence $lit(c_1)$ and $lit(c_2)$ must be the watched literals for Cl. To check if Cl appears in the SAT solver already, we check all clauses where $lit(c_1)$ is a watched literal (the SAT solver provides this data structure), and see if one is identical to Cl. This check is reasonably expensive, but much cheaper than looking at all clauses involving $lit(c_1)$ since it will be the watched literal in few of them.

9 Experiments

We have built a prototype lazy clause generator system using MiniSat [26] version 2.0 beta as the starting point. We now give the results of a number of different experiments using lazy clause generation. We also compare various modelling choices for lazy clause generation on some problems. We use 3 letter codes to define a modelling choice: the first letter indicates **(e)**ager or **(l)**azy modelling; the second letter indicates **(f)**ull integer representation, **(n)**on-continuous representation, **(b)**ounds representation, and n**(o)**n-continuous bounds representation; and the third letter represents **(b)**ounds or **(d)**omain propagators. Note that for the eager approach with bounds variable representation the clauses for the bounds and domain propagators are exactly the same, and thus we write **eb(bd)** to denote **ebb** and **ebd**. We compare our approach versus eager approaches using the MiniSat [26] version 2.0 beta as the SAT solver, and versus the Gecode 1.3.1 [16] finite domain propagation system.

The open-shop scheduling experiments were run on a 3 GHz Intel Pentium D with 4 Gb RAM running Debian Linux 3.1, while the remaining experiments were run on a 3.4 GHz Intel Pentium D with 4 Gb RAM running Debian Linux 4.

9.1 Open shop scheduling problems

The first set of experiments use open-shop scheduling problems from [11]. Each of the constraints in these problems is of the form $x_1 \vee x_2, x_1 + d \leqslant x_2$ or $x_0 \Leftrightarrow x_1 + d \leqslant x_2$ where d is a constant. These problems are also amenable to solving using SAT modulo difference logic. All of the propagators we use are tight bounds propagators so we only use Boolean variables of the form $[\![x \leqslant d]\!]$ and the first class of clause for $DOM(x)$. We use eager models for the first two kinds of constraints $x_1 \vee x_2, x_1 + d \leqslant x_2$ since they can be modelled with a linear (in domain size) number of binary clauses. We use lazy propagators for the reified difference inequalities $x_0 \Leftrightarrow x_1 + d \leqslant x_2$.

We compare our lazy clause generation approach versus the eager modelling approach where we used the minimal clausal representations generated by [33]. The eager models are run with MiniSat version 2.0 beta as the SAT solver. For open-shop scheduling problems we do not compare against Gecode, because without very sophisticated encodings and search strategies [22], they are not competitive on these problems, since they lack nogoods. Instead we compare against the Barcelogic DPLL(T) SAT modulo theories (SMT) solver version 1.1 using its difference logic theory solver [3], since these problems fall into the class of difference logic problems which can be handled by this solver very efficiently.

These scheduling problems are optimization problems. we search for the minimal makespan (completion time for all jobs). The minimization is conducted by dichotomic search over the space of possible makespans, see [33] for details. We note that dichotomic optimization search is in a sense advantageous to the eager modelling approach since it generates clauses once which are effectively used in solving multiple (linked) satisfaction sub-problems.

Since these are large suites of benchmarks, we show summary results as well as a few individual instances to illustrate the spread of results. In each table we show the user time to find and prove the optimal solution for: the lazy approach **lbb**, the eager approach **ebb** (and just the time spent in the SAT solver for the eager approach **sat**), and the SMT approach **smt**. We also give the number of conflicts for each approach,

Table 1 Open shop scheduling suite `gp` (80 instances)

Benchmark	Time (s)				Conflict number			Clause ratio	
	lbb	**ebb**	**sat**	**smt**	**lbb**	**ebb**	**smt**	ave	min
gp04-09	0.38	6.84	1.31	0.17	32	21	39	5.15	5.15
gp05-01	1.41	27.32	6.53	0.27	39	19	61	5.67	5.67
gp08-09	5.09	136.62	32.25	0.86	129	53	121	9.05	9.05
gp10-07	16.25	347.60	99.30	9.53	622	622	1400	11.05	10.97
gp10-10	21.68	410.34	115.79	7.80	995	857	1371	10.85	10.82
Arith. mean	6.04	113.46	30.05	2.59	311	242	492	7.43	7.40
Geom. mean	2.49	47.43	11.14	0.59	100	48	94	7.03	7.02

and the average and minimum, across all sub-problems in the dichotomic search, of the ratio of clauses for the eager approach divided by the total created by the lazy approach.

The open-shop scheduling suite `gp` shown in Table 1 is easy for all approaches. For these problems **ebb** spends most of its time just generating the clauses. While clearly **smt** requires more search to find the solution, given the tiny description of the problem for **smt** it is very rapid. Note that some of these problems were only closed in 2005 [22], so they are not considered easy for technologies without nogoods. Indeed tackling `gp06-*` problems in Gecode fails to find a solution within 5 minutes.

The open-shop scheduling suite `tai` shown in Table 2 is more difficult. As the problem size grows the advantage of the lazy and eager approaches grows over the SMT approach. The search space explored by the lazy approach is around twice that of the eager approach, but it is still uniformly faster. Note also that the larger the example the smaller the percentage of clauses generated by the lazy approach.

The open-shop scheduling suite `j` shown in Table 3 is much harder. The three hardest problems `j7-per0-0`, `j8-per0-1`, and `j8-per10-2` which were closed recently [33] are examined separately. The lazy approach is better than the eager approach except for `j7-per10-2`, and better than SMT on the larger problems. To save experimental time for the three hardest problems we only try to find a solution with optimal makespan (a single sub-problem) (dichotomic search for the largest problem takes over 2 days for **ebb**). Surprisingly **ebb** improves on **lbb** for two of these problems, showing that having all the clause information from the beginning can be advantageous. The main extra cost appears to be the size of nogoods generated.

Table 2 Open shop scheduling suite `tai` (60 instances)

Benchmark	Time (s)				Conflict number			Clause ratio	
	lbb	**ebb**	**sat**	**smt**	**lbb**	**ebb**	**smt**	ave	min
tai_5x5_1	0.42	4.64	1.08	0.95	887	774	1679	6.33	5.53
tai_7x7_6	16.23	23.75	10.37	452.15	12722	4397	264167	7.38	5.38
tai_10x10_1	7.52	78.76	18.65	674.99	3614	1599	108764	12.90	10.63
tai_10x10_10	3.80	79.32	17.97	33.34	1431	2675	7848	13.21	12.66
tai_20x20_4	269.89	1361.31	369.42	601.35	11247	3782	39831	26.23	24.42
tai_20x20_8	424.78	1420.77	428.60	6035.09	56092	15891	345876	24.42	20.51
Arith. mean	62.42	317.95	88.39	631.78	6611	3597	43231	13.17	12.03
Geom. mean	4.02	42.47	9.98	21.12	1783	1231	5565	11.20	10.14

Table 3 Open shop scheduling suite `j` (48 + 3 instances)

Benchmark	Time (s)				Conflict number			Cl. ratio	
	lbb	**ebb**	**sat**	**smt**	**lbb**	**ebb**	**smt**	ave	min
j3-per0-2	0.29	4.02	0.89	0.15	57	20	31	3.46	3.46
j6-per0-0	500.68	703.66	638.23	277.67	158117	137911	212512	6.22	5.35
j7-per10-1	25.45	84.75	36.84	47.83	8967	5019	23478	8.23	7.75
j7-per10-2	1451.79	1437.52	1379.42	3136.69	303011	250942	1625354	6.90	5.04
j8-per20-0	19.02	104.56	36.57	552.40	5493	3138	186300	9.36	8.55
Arith. mean	113.48	252.97	226.08	298.71	25430	29877	110525	6.51	6.21
Geom. mean	3.19	29.37	8.96	2.66	780	559	937	6.30	6.04
j7-per0-0-sat	8443	5246	5210	11470	991907	533852	4328222	3.92	3.92
j8-per0-1-sat	19031	34322	34246	32413	1828054	1452649	8539727	5.90	5.90
j8-per10-2-sat	2205	1395	1322	3846	209822	160075	1316112	5.52	5.52

Overall, **lbb** solves faster than **ebb** except for `j7-per0-0`, `j8-per10-2` and `j7-per10-2`. Across the suites it is an order of magnitude faster in geometric mean. While it requires more search than **ebb**, the massive reduction in clauses pays off. The lowest clause ratio that occurs in any instance is 3.46. Overall **lbb** generally improves upon **smt** the harder the examples become.

9.2 Crypt-arithmetic problems

The next set of problems are crypt-arithmetic problems like the famous: SEND+MORE=MONEY problem where each letter represents a different digit and the equation has to hold. They involve large linear equations and a single `alldifferent` constraint. For none of these problems could the eager approach [33] generate the clauses within hours.

All the models use bounds propagators for the large linear equation, while the third code letter represents the style of propagator used for the `alldifferent`. The lazy clause generation approach uses an **alldifferent** propagator equivalent to propagation on a set of independent disequations ($x_1 \neq x_2$) using either domain propagators or bounds propagation for the disequations, while Gecode uses its native `distinct` propagator. All solvers look for all solutions. We compare **lfd**, **lbd** and **lbb** using VSIDS search and first fail search versus Gecode using first fail search.

We compare the approaches on the well known `alpha` problem and instances taken from [9]. We show the instances from [9] which require more than 1000 conflicts/failure for some solver and search. A full description of the problems can be found at [4].

The results are shown in Tables 4 and 5. Clearly the flexibility of using domain propagators on a bounds representation can be beneficial since **lbd** outperforms **lfd**, but the best lazy approach is simply using bounds propagation. Clearly nogoods can significantly reduce the search for these problems. The highly engineered Gecode solver propagates much faster than our naive propagation engine, and there is not enough search here to really benefit from nogoods. Interestingly here is a case where VSIDS search is bettered by a more usual CP style search, although admittedly the problems are easy.

Table 4 Crypt-arithmetic problems: user-time

Benchmark	VSIDS			First fail			
	lfd	**lbd**	**lbb**	**lfd**	**lbd**	**lbb**	gecode
alpha	0.02	0.01	0.001	0.01	0.01	0.01	0.001
problem0	0.62	0.30	0.34	0.39	0.28	0.32	0.04
problem1	0.17	0.17	0.25	0.11	0.10	0.11	0.02
problem2	0.16	0.18	0.15	0.18	0.18	0.15	0.03
problem3	1.12	0.79	0.69	0.13	0.15	0.14	0.02
problem4	0.03	0.03	0.05	0.001	0.001	0.01	0.001
problem6	0.37	0.30	0.36	0.19	0.20	0.17	0.02
problem34	0.25	0.34	0.03	0.18	0.001	0.001	0.001
problem57	0.12	0.11	0.72	0.001	0.03	0.03	0.001
problem63	0.61	0.31	0.10	0.03	0.02	0.02	0.01
Arith mean	0.38	0.35	0.32	0.11	0.07	0.07	0.01
Geom mean	0.24	0.20	0.16	0.05	0.04	0.03	0.01

9.3 Quasigroup completion problems

A $n \times n$ *latin square* is a square of values x_{ij}, $1 \leqslant i, j \leqslant n$ where each number $[\,1\,..\,n\,]$ appears exactly once in each row and column. It is represented by constraints

$$\texttt{alldifferent}([x_{i1}, \ldots, x_{in}],\ 1 \leqslant i \leqslant n$$
$$\texttt{alldifferent}([x_{1j}, \ldots, x_{nj}],\ 1 \leqslant j \leqslant n$$

The quasigroup completion problem (QCP) is a latin square problem where some of the x_{ij} are given. These are challenging problems which exhibit phase transition behaviour. We use examples from the 2006 Constraint Satisfaction Solver Competition [10], and some larger examples generated by the `lsencode` [23] generator. Again the propagators for the `alldifferent` constraint are equivalent to propagation on a set of independent disequations ($x_1 \neq x_2$) using either domain propagators or bounds propagators for the disequations. We compare against Gecode using `distinct` with first fail search.

Table 5 Crypt-arithmetic problems: conflicts

Benchmark	VSIDS			First fail			
	lfd	**lbd**	**lbb**	**lfd**	**lbd**	**lbb**	gecode
alpha	41	47	51	34	36	34	33
problem0	10025	6535	7201	6743	6280	7213	8213
problem1	3718	4203	6213	2579	2479	2746	4008
problem2	3559	4274	4150	3986	3998	3519	6204
problem3	15137	13124	12089	2511	2647	2543	2560
problem4	725	789	1396	182	195	183	181
problem6	6271	6335	7898	3444	3533	3439	3737
problem34	4861	7616	789	3999	200	164	62
problem57	2647	2950	12287	112	529	535	337
problem63	9122	7330	2737	604	488	468	427
Arith mean	6077	6200	6195	2035	1592	1507	1691
Geom mean	4275	4265	4058	1089	899	815	825

Table 6 QCP 15 × 15 instances: user time

Benchmark	Time (s)					
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**	gecode
qcp-15-120-0_ext	0.03	0.02	0.03	0.14	0.02	0.02
qcp-15-120-1_ext	0.03	0.04	0.06	0.22	0.04	0.08
qcp-15-120-2_ext	0.06	0.02	0.05	0.16	0.03	454.53
qcp-15-120-3_ext	0.03	0.04	0.14	0.26	0.03	0.19
qcp-15-120-4_ext	0.18	0.02	0.02	0.33	0.22	5.50
qcp-15-120-5_ext	0.13	0.09	0.21	0.62	0.15	117.08
qcp-15-120-6_ext	0.02	0.02	0.01	0.17	0.04	38.01
qcp-15-120-7_ext	0.10	0.13	0.29	0.24	0.39	1.28
qcp-15-120-8_ext	0.04	0.10	0.04	0.18	0.06	6.70
qcp-15-120-9_ext	0.06	0.14	0.24	0.27	0.07	1685.44
qcp-15-120-10_ext	0.04	0.04	0.04	0.20	0.04	1044.80
qcp-15-120-11_ext	0.01	0.05	0.01	0.32	0.05	47.64
qcp-15-120-12_ext	0.01	0.01	0.02	0.04	0.01	862.29
qcp-15-120-13_ext	0.14	0.30	0.17	0.21	0.46	179.18
qcp-15-120-14_ext	0.01	0.01	0.01	0.01	0.12	2034.72
Arith mean	0.08	0.07	0.09	0.22	0.12	431.83
Geom mean	0.06	0.04	0.05	0.17	0.07	24.67

Tables 6 and 7 compare the user time and amount of search for finding the first solution of quasigroup completion problems of size 15 × 15 for various modelling possibilities. For eager modelling the time for constructing the clausal representation is included, it is either 0.01 or 0.02 seconds. The benchmarks 0–9 are satisfiable while 10–14 are unsatisfiable.

We can see that nogoods are really effective in these problems in reducing search. While Gecode is much more efficient in propagation, for hard examples the reduction

Table 7 QCP 15 × 15 instances: conflicts/failures

Benchmark	Conflicts/Failures					
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**	gecode
qcp-15-120-0_ext	369	174	151	1437	148	301
qcp-15-120-1_ext	312	635	522	2113	211	2717
qcp-15-120-2_ext	1214	455	599	1886	468	15994711
qcp-15-120-3_ext	470	767	1319	2961	223	6195
qcp-15-120-4_ext	2823	156	88	3870	2595	196310
qcp-15-120-5_ext	1995	1354	1964	6106	1520	3892358
qcp-15-120-6_ext	271	148	59	1615	326	1334910
qcp-15-120-7_ext	1259	2091	2440	2569	3942	43141
qcp-15-120-8_ext	458	1771	401	2028	715	226380
qcp-15-120-9_ext	1045	2199	2376	3163	890	59032321
qcp-15-120-10_ext	762	805	497	2453	650	38584120
qcp-15-120-11_ext	19	843	27	3529	382	1619456
qcp-15-120-12_ext	49	85	164	556	97	32919163
qcp-15-120-13_ext	2172	4437	1578	2349	5077	6364807
qcp-15-120-14_ext	26	45	42	157	2041	74004918
Arith mean	883	1064	815	2453	1286	15614787
Geom mean	422	546	359	1928	673	814626

Table 8 QCP 25 × 25

Benchmark	Time (s)					Conflicts ('000)				
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**	**efd**	**eb(bd)**	**lfd**	**lbd**	**lbb**
qcp-25-264-0_ext	114.07	65.56	149.88	85.89	34.81	212	117	159	174	98
qcp-25-264-1_ext	832.31	108.37	99.84	374.77	258.95	1037	178	119	626	608
qcp-25-264-2_ext	15.40	44.40	12.25	47.34	41.34	44	99	29	125	146
qcp-25-264-3_ext	542.61	273.36	442.57	532.47	471.86	814	393	399	892	1123
qcp-25-264-4_ext	265.00	268.84	24.87	418.33	65.55	417	405	42	760	193
qcp-25-264-5_ext	108.60	146.36	341.25	158.62	311.52	210	256	325	345	790
qcp-25-264-6_ext	255.60	185.53	130.06	127.91	67.32	397	282	161	273	185
qcp-25-264-7_ext	35.36	1.52	34.07	78.26	94.09	84	9.6	60	178	238
qcp-25-264-8_ext	9.52	48.36	81.10	171.35	137.44	30	96	102	352	360
qcp-25-264-9_ext	27.80	153.52	286.20	710.96	49.41	70	261	291	1301	155
qcp-25-264-10_ext	30.92	125.67	165.77	346.78	415.15	76	226	182	709	1058
qcp-25-264-11_ext	0.14	0.06	0.10	0.17	0.05	0.2	0.2	0.3	0.7	0.2
qcp-25-264-12_ext	0.23	0.21	0.24	0.32	0.16	1.6	3.1	2.1	2.8	1.3
qcp-25-264-13_ext	0.36	0.29	0.34	0.34	0.69	4.1	4.1	3.9	3.1	6.0
qcp-25-264-14_ext	107.82	131.88	175.01	176.97	58.41	192	208	170	352	168
Arith mean	156.38	103.60	129.57	215.37	133.78	239	169	136	406	342
Geom mean	26.40	23.75	30.31	53.07	30.74	64	61	53	137	100

in search space is so dramatic that it cannot compete. The eager approaches are best for these examples, while the **lfd** combination is the best lazy approach. This is interesting as the bounds propagation is worse than domain propagation for the lazy approach, but better for the eager approach.

Table 8 shows the results on 25 × 25 QCP problems in order to see the trend for modelling choices as size increases. These problems are hard for Gecode, taking hours to complete. In 8 out of 15 instances **lfd** improves upon the eager approach **efd**, and overall it solves the whole suite faster. Even though QCP problems are small (the cost of eager clause generation is less than 0.10 seconds) the lazy approach avoids the overhead of examining many useless clauses, and hence starts outperforming the eager approach as the problem size grows. Interestingly **eb(bd)** is still better than the lazy approach **lfd** for these problems, even though the lazy bounds representations are poorer than **lfd**. Examining the novel combination **lbd** where it has the same search as **lfd** it is substantially faster, but usually the search space is bigger, and even bigger than the weaker **lbb** strangely.

In order to further view the trends as the problem size increased we generated problems of size 35 × 35 with 600 holes [4] using the `lsencode` [23] problem generator. Results are presented in Table 9, where all problems are satisfiable. We can see that now **lfd** is the best method overall, significantly beating the eager version **efd**, and just better than the eager bounds version **eb(bd)**. For these larger problems **lbd** now improves upon **lbb** and actually gives the best results for 5 out of 20 instances. With a better implementation of duplicate checking it might be quite competitive.

This shows that it may well be the case that lazy propagation is worthwhile even for problems where an eager encoding is quite good. One should remember that there are very few constraints in the propagation engine, QCP $n \times n$ requires only $2n$ `alldifferent` constraints.

Table 9 QCP 35 × 35

Benchmark	Time (s)					Conflicts ('000)				
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**	**efd**	**eb(bd)**	**lfd**	**lbd**	**lbb**
qcp35-600-0	4299	474	741	202	1612	2326	436	642	388	1390
qcp35-600-1	919	1323	226	204	231	653	797	286	293	437
qcp35-600-2	164	424	283	107	136	392	628	403	290	388
qcp35-600-3	1976	309	1641	1316	433	1132	288	904	1075	471
qcp35-600-4	954	451	23	214	639	783	431	94	410	752
qcp35-600-5	1514	5	1431	446	2589	1256	30	809	868	2155
qcp35-600-6	1850	1584	654	1868	129	1709	901	524	1259	227
qcp35-600-7	100	97	771	4413	299	227	260	813	3275	557
qcp35-600-8	56	144	184	200	91	142	248	210	365	181
qcp35-600-9	125	70	19	179	1380	366	250	85	520	1214
qcp35-600-10	1932	1167	782	917	288	1609	761	779	844	796
qcp35-600-11	168	2744	79	97	687	286	1242	189	230	989
qcp35-600-12	770	130	912	478	1665	628	235	719	1145	1613
qcp35-600-13	439	20	237	374	409	473	55	278	565	467
qcp35-600-14	156	1446	724	1562	2907	344	1128	532	1333	2914
qcp35-600-15	3883	33	582	86	655	1563	69	644	317	761
qcp35-600-16	817	1775	342	126	846	570	965	352	334	853
qcp35-600-17	233	784	1322	50	547	339	716	773	167	693
qcp35-600-18	19	216	2122	1398	429	80	365	1599	2002	560
qcp35-600-19	2084	200	114	1047	12	1123	314	181	1075	57
Arith mean	1123	670	660	764	799	800	506	541	838	874
Geom mean	506	277	373	376	448	578	353	421	617	638

9.4 CELAR radio link frequency assignment problems

The CELAR Radio Link Frequency Assignment Problems [5] consist of a set of radio frequencies and a set of radio links to assign a frequency to each radio link. Some pairs of radio links must be an exact distance apart in frequency, while other should be at least some distance apart. We use the first 5 problems (where all constraints are mutually satisfiable) while minimizing the maximum frequency used. The set of possible frequencies F is non-continuous:

$$\{2 + 14i | 1 \leqslant i \leqslant 11\} \cup \{2 + 14i | 18 \leqslant i \leqslant 28\}$$
$$\cup \{8 + 14i | 29 \leqslant i \leqslant 30\} \cup \{8 + 14i | 46 \leqslant i \leqslant 56\},$$

using only 44 values in the range [16 .. 792] of 777 possible values. So these problems are candidates for the non-continuous representation. We model the problem using

Table 10 CELAR problems: user time

Prob	User time (s)				
	lfb	**lnb**	**lbb**	**lob**	gecode
scen01	285.22	13.67	104.65	9.37	>400
scen02	2.03	0.16	0.86	0.11	>400
scen03	39.90	3.16	20.06	2.19	>400
scen04	2.17	0.16	0.88	0.10	0.46
scen05	2.25	0.17	0.96	0.10	0.34

Table 11 CELAR problems: Conflicts/Failures

Prob	Conflicts/Failures				
	lfb	**lnb**	**lbb**	**lob**	gecode
scen01	5036	4542	4160	4247	–
scen02	202	127	180	261	–
scen03	3039	2380	2667	2553	–
scen04	7	6	2	1	31
scen05	17	22	36	24	74

bounds propagators for $|x - y| \geqslant k$ (see Example 24), and model $|x - y| = k$ using the bounds propagators for the individual constraints $|x - y| \geqslant k$, $x - y \leqslant k$ and $y - x \leqslant k$.

We compare the **f**ull integer representation, **n**on-continuous representation, **b**ounds representation, and n**o**n-continuous bounds representation. For the full integer representation we statically add constraints $\neg[\![x = d]\!]$, $d \in [\,16\,..\,792\,] - F$ to the SAT solver, while for the (continuous) bounds representation we statically add the constraints $\neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i+1}]\!]$ where d_i and d_{i+1} are consecutive values in F. We also compare with Gecode using reified constraints to represent $|x - y| \geqslant k$ as $x - y \geqslant k \vee y - x \geqslant k$.

The results for the various modelling choices are shown for: user time in Table 10, failures in Table 11, and unit propagation executed in Table 12. Clearly the non-continuous representations are significantly better than the continuous representations, they involve around 20× fewer variables. The failure results show that it is not the results of a better search because there are fewer Boolean variables to branch on, instead it is simply the overhead of more unit propagations to deal with the larger number of variables.

This clearly shows the benefit of separation of propagator implementation from variable representation. The propagator is highly effective on the non-continuous Boolean representations without being modified.

Interestingly for these problems the disjunctive propagator explained in Example 24 does not improve upon the bounds propagator.

9.5 Lazy clauses as nogoods

The lazy clause generation approach adds clauses representing the propagators to the SAT solver permanently, so they can never be removed. But since we continually run the propagation engine, this is not necessary. If we removed them later they would be rediscovered by the propagation engine, if needed. It seems worthwhile then to

Table 12 CELAR problems: unit propagations

Prob	Unit Propagations			
	lfb	**lnb**	**lbb**	**lob**
scen01	177561515	13081789	133403108	7133763
scen02	1969516	183084	1732660	112612
scen03	43087573	3608960	38598246	1918102
scen04	628192	36289	304949	17368
scen05	901257	65516	1375927	47145

Table 13 Treating lazy clauses as nogoods

Suite	Time	Conflicts	Lazy Clauses
gp	+11.2%	+4.9%	+65.7%
tai	+43.4%	+26.7%	+53.5%
qcp-25	+20.5%	+20.5%	+3.6%

consider treating these added clauses like nogood clauses, which are added and then later removed when they seem not be useful.

The final experiment checks if giving lazy clauses as nogoods could improve an overall performance of our solver, i.e. leaving it to a SAT solver to decide which lazy clauses to keep. We changed our implementation slightly to give a lazy clause as follows. We do not add a conflict clause to the clause store of the SAT solver but simply give it as a reason for a conflict. A clause with an implied literal is added as a learnt clause.

We compared either keeping clauses from propagators as permanent clauses or as nogoods on the gp and tai open-shop scheduling suites using nlbb and QCP 25×25 suite using **lfd**. The relative performance of the version with lazy clauses treated as nogoods is shown in Table 13. The table illustrates that it is not beneficial to keep lazy clauses as nogoods, and doing so always increased the search and the number of clauses generated.

Note that we only used MiniSATs default strategy for nogood management, which may not be suited to the lazy clause generation, so this should probably be more deeply investigated. Clearly if the size of the lazy clauses generates becomes prohibitive for some problems the cost of using nogoods to store lazy clauses is not prohibitive in any case.

10 Related work

The paper [33] explains how to eagerly encode linear arithmetic constraints into CNF (to give tight clauses) using the propositions $[\![x \leqslant d]\!]$. They closed three very hard open-shop scheduling problems using their eager approach, but the approach is manifestly impractical when the linear constraint involves a significant number of variables. Our lazy approach makes the encoding of linear arithmetic possible for large linear constraints, and allows encoding of arbitrary propagators.

Gent [15] describes how to encode arc consistency in SAT using the *support encoding* [18]. This is tantamount to encoding the propagation rules of an arc consistency algorithm. Gent [15] shows that the support encoding is more efficient than the usual direct encoding of binary CSPs to SAT. Again the approach, which is analogous to that of [33], is completely impractical for large arity constraints.

The closest related work to this paper is the hybrid BDD and SAT bounds propagation set solver described in [17]. There a BDD-based set solver and a SAT solver are integrated and the BDD set solver passes clauses describing its propagations to the SAT solver in order to make use of the nogood capabilities of the SAT solver. Using BDD propagators, the construction of tight propagation rules can be automatic. Here we extend the approach beyond set variables to support integer variables, eliminate the propagation solver by embedding the minimal amount of machinery required into the SAT solver.

There is a substantial body of work on look back methods in constraint satisfaction (see e.g. [13], chapter 6), but there was little evidence until recently of success for look back methods that combine with propagation. The work of Katsirelos and Bacchus [19] showed that one could use nogood technology derived from SAT for storing and managing nogoods in a CSP system using FC-CBJ. In further work [20] they consider how to generate explanations (which are effectively clauses) of propagation for a number of global constraints, in order to support nogoods in a CP solver. They consider the usual DIMACS encoding of integers $\{[\![x = d]\!]\}$ and hence do not consider bounds propagation.

Roussel [31] gave a linear encoding of domains (not including inequality literals) which has the same unit propagation strength as our new encoding, but requires more variables and literals.

The lazy propagation approach can be viewed as a special form of Satisfiability Modulo Theories [28] solver, where each propagator is considered as a separate theory, and theory propagation is used to learn clauses.

There are other propagation solvers which allow different representation of integers, in particular Minion [25] and Gecode [16]. All representations either support all atomic constraints or are restricted in the propagators they can be used. The views approach of Gecode [32] allows variables defined by simple constraints to be seen as mappings from atomic constraint to atomic constraints, and hence has some similarity with the mapping idea of this paper. For example a variable $y = x + 3$ effectively rewrites atomic constraint like $x \geqslant 4$ to $y \geqslant 6$ and vice versa. It would be useful to include views in the lazy clause generation solver, since it reduces the number of Boolean variables required.

11 Conclusion

In conclusion, we have constructed a hybrid SAT finite domain propagation solver using lazy clause generation that captures some of the advantages of both paradigms. It can tackle hard scheduling problems efficiently without complex search strategies. Where large amounts of search are required we expect it to be more effective than propagation based solvers because it includes nogoods and conflict directed backjumping. We have examined the modelling choices that arise from the lazy clause generation hybrid solving approach. We find that the separation of choice of propagator from Boolean variable representation leads to an increased number of modelling choices. The direct representation of non-continuous variables is clearly advantageous, and there is some evidence that the use of disjunctive propagators (domain propagators for bounds variables) can improve upon other modelling approaches. It also appears there sometimes even if an eager model is quite small, as in QCP, it still may be preferable to use a lazy propagation approach. But we have only really scratched the surface of the possibilities of the lazy approach.

References

1. Ansótegui, C., & Manyá, F. (2004). Mapping problems with finite-domain variables into problems with Boolean variables. In *Proceedings of the seventh international conference on theory and applications of satisfiability testing (SAT'04)*. *LNCS* (Vol. 3542, pp. 1–15).

2. Bailleux, O., & Boufkhad, Y. (2003). Efficient CNF encoding of Boolean cardinality constraints. In F. Rossi (Ed.), *Proceedings of the 9th international conference on principles and practice of constraint programming (CP2003)*. *LNCS* (Vol. 2833, pp. 108–122).
3. Barcelogic for SMT. www.lsi.upc.es/∼oliveras/bclt-main.html. Accessed 07 February.
4. Benchmarks for Lazy Clause Generation. http://www.cs.mu.oz.au/~olgao/benchmarks.htm. Accessed 07 December.
5. Cabon, B., de Givrey, S., Lobjois, L., Schiex, T., & Warners, L.P. (1999). Radio link frequency assignment. *Constraints, 4*(1), 78–89.
6. Choi, C.W., Lee, J.H.M., & Stuckey, P.J. (2003). Propagation redundancy in redundant modelling. In F. Rossi (Ed.), *Proceedings of the ninth international conference on principles and practices of constraint programming (CP2003)*. *LNCS* (Vol. 2833, pp. 229–243).
7. Choi, C.W., Lee, J.H.M., & Stuckey, P.J. (2007). Removing propagation redundant constraints in redundant modeling. *ACM Transactions on Computational Logic, 8*(4), article 23.
8. Crawford, J., & Baker, A. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th national conference on artificial intelligence (AAAI'94)* (pp. 1092–1097).
9. Cryptarithmetic puzzles. http://www.tkcs-collins.com/truman/alphamet/alphamet.shtml. Accessed 07 December.
10. CSP competition (2006). http://cpai.ucc.ie/06/Competition.html. Accessed 07 June.
11. CSP2SAT. http://bach.istc.kobe-u.ac.jp/csp2sat/. 06 December.
12. Davis, M., Logemman, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM, 5*(7), 394–397.
13. Dechter, R. (2003). *Constraint processing*. San Francisco: Morgan Kaufmann.
14. Eén, N., & Sörensson, N. (2006). Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation, 2*, 1–26.
15. Gent, I. P. (2002). Arc consistency in SAT. In *Proceedings of the 15th Eureopean conference on artificial intelligence, ECAI'2002, Lyon, France, July 2002* (pp. 121–125).
16. GECODE. www.gecode.org. Accessed 07 February.
17. Hawkins, P., & Stuckey, P.J. (2006). A hybrid BDD and SAT finite domain constraint solver. In P. Van Hentenryck (Ed.), *Proceedings of the practical applications of declarative programming (PADL'06)*. *LNCS* (Vol. 3819, pp. 103–117).
18. Kasif, S. (1990). On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence, 45*, 275–286.
19. Katsirelos, G., & Bacchus, F. (2003). Unrestricted nogood recording in CSP search. In F. Rossi (Ed.), *Proceedings of the 9th international conference on principles and practice of constraint programming (CP2003)*. *LNCS* (Vol. 2833, pp. 873–877).
20. Katsirelos, G., & Bacchus, F. (2005). Generalized nogoods in CSPs. In *The twentieth national conference on artificial intelligence (AAAI'05)* (pp. 390–396).
21. Kautz, H.A., & Selman, B. (1992). Planning as satisfiability. In *Proceedings of the tenth European conference on artificial intelligence (ECAI'92)* (pp. 359–363).
22. Laborie, P. (2005). Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings of the nineteenth international joint conference on artificial intelligence (IJCAI'05)* (pp. 181–186).
23. Lsencode. http://www.cs.cornell.edu/gomes/SOFT/lsencode-v1.1.tar.Z/. Accessed 07 November.
24. Marriott, K., & Stuckey, P.J. (1998). *Programming with constraints: An introduction*. Cambridge: MIT.
25. Minion. minion.sourceforge.net. Accessed 07 Feb.
26. MiniSat. www.cs.chalmers.se/Cs/Resarch/FormalMethods/MiniSat/. Accessed 06 December.
27. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of 38th conference on design automation (DAC'01)* (pp. 530–535).
28. Niewenhuis, R., Oliveras, A., & Tinelli, C. (2004). Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the 11th international conference on logic for programming artificial intelligence and reasoning (LPAR'04)*. *LNAI* (Vol. 3452, pp. 36–50)
29. Ohrimenko, O., & Stuckey, P.J. (2008). Modelling for lazy clause generation. In J. Harland, & P. Manyem (Eds.), *Proceedings of the fourteenth computing: The Australasian theory symposium (CATS 2008)*. *CRPIT* (Vol. 77, pp. 27–38)
30. Ohrimenko, O., Stuckey, P.J., & Codish, M. (2007). Propagation = lazy clause generation. In C. Bessiere (Ed.), *Proceedings of the 13th international conference on principles and practice of constraint programming*. *LNCS* (Vol. 4741, pp. 544–558)

31. Roussel, O. (2005). Some notes on the implementation of csp2sat+zchaff, a sim- ple translator from CSP to SAT. In *Proceedings of the 2nd international workshop on constraint propagation and implementation* (pp. 83–88).
32. Schulte, C., & Tack, G. (2005). Views and iterators for generic constraint implementations. In P. van Beek (Ed.), *Proceedings of the 11th international conference on principles and practice of constraint programming (CP 2005). Lecture notes in computer science* (Vol. 3709, pp. 817–821).
33. Tamura, N., Taga, A., Kitagawa, S., Banbara, M. (2006). Compiling finite linear CSP to SAT. In F. Benhamou (Ed.), *Proceedings of 12th international conference on principles and practice of constraint programming (CP2006). LNCS* (Vol. 4204, pp. 590–603).
34. Van Hentenryck, P., Saraswat, V., & Deville, Y. (1998). Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming, 37*(1–3), 139–164.
35. Walsh, T. (2000). SAT v CSP. In R. Dechter (Ed.), *Proceedings of 6th international conference on principles and practice of constraint programming (CP2000). LNCS* (Vol. 1894, pp. 441–456).