Propagation = Lazy Clause Generation

Olga Ohrimenko[1], Peter J. Stuckey[1], and Michael Codish[2]

[1] NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng.
University of Melbourne, Australia
[2] Department of Computer Science, Ben-Gurion University, Israel

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 we can 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 we can convert propagators to lazy clause generators for a SAT solver. The resulting system can solve scheduling problems significantly faster than generating the clauses from scratch, or using Satisfiability Modulo Theories solvers with difference logic.

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 tradeoffs between the speed of inference of information versus the strength of the information inferred. Good propagators represent a good tradeoff at least for some problem classes. The success of finite domain propagation in solving hard combinatorial probles arises from these good tradeoffs, 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. 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 [13].

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 uncovers an Achilles' heel of SAT solvers. While modern SAT solvers can often handle problems with millions of constraints 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.

This paper contributes a hybrid system for implementing propagation-based finite domain solving with a SAT solver and demonstrates its successful application to hard open-shop scheduling benchmarks. We compare the hybrid solver with a static approach that introduces the propagation clauses a priori, and with Satisfiability Modulo Theories (SMT) [14] solving using difference logic. Our prototype implementation can significantly improve on the carefully engineered SAT and SMT solvers.

In the remainder of the paper we first introduce terminology, and then propagation rules, a method of understanding propagator behaviour. We show how these can be expressed as CNF formulae, and introduce the lazy clause generation approach. After experiments we compare with related work and 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. 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 defined by the domain $D_1(v) \cap D_2(v)$ for all $v \in \mathcal{V}$.

We use *range* notation: For integers l and u, $[\,l\,..\,u\,]$ 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 *convex* domain D is where $D(T)$ 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 $v \in \mathit{vars}(\theta)$.

A constraint is a restriction placed on the allowable 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. $\mathit{solns}(c) = \{\theta \mid (\mathit{vars}(\theta) = \mathit{vars}(c)) \ \wedge \ (\vDash \theta(c))\}$

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

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

Example 1. 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 $D_1(x_1) = \{3, 4, 6, 8\}$ and $D_1(x_2) = \{1, 5\}$, then $f(D_1)(x_1) = \{3, 4\}$ and $f(D_1)(x_2) = \{1, 5\}$. □

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

$$\mathit{solv}(F, D) = \mathrm{gfp}(\lambda d.\mathit{iter}(F, d))(D) \qquad \mathit{iter}(F, D) = \bigsqcap_{f \in F} f(D).$$

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, $\neg l$ 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. An *assignment* is either a set of literals A excluding $\bot$ such that $\forall p \in \mathcal{P}.\{p, \neg p\} \not\subseteq A$, or the failed assignment $\overline{\bot}$. We define $\overline{\bot} \cup A = \overline{\bot}$ for any assignment A.

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 DPLL algorithm 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. It can be defined as follows where C denotes a clause:

$$up(A,C) = \begin{cases} \overline{\bot} & \forall l \in C. \neg l \in A \\ A \cup \{l\} & \exists l \in C, , \neg l \notin A, \forall l' \in (C \setminus \{l\}). \neg l' \in A \\ A & \text{otherwise} \end{cases}$$

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

4 Atomic Constraints and Propagation Rules

Atomic constraints and propagation rules were originally devised for reasoning about propagation redundancy [1]. 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 unsatisfiabity 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.

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

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 2. A common propagator f for the constraint $x_1 = x_2 \times x_3$ [11] is

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

and symmetrically for x_3.[3] Note that f does not enforce any notion of consistency.

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

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

Let $rules(f)$ be the set of all possible propagation rules implemented by f. This definition of $rules(f)$ is usually unreasonably large, and full of redundancy. For example the second propagation rule in Example 2 is clearly weaker than the first.

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

In order to reason more effectively about propagation rules for a given propagator f, we want to have a concise representation $rep(f)$ such that $rep(f)$ implements f.

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.

Example 3. 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

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

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. □

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.

[3] Division by zero has to be treated carefully here, see [11] for details.

5.1 Atomic constraints and Boolean variables

Changes in domains of variables are the information recorded by a propagation solver. In this sense they are the "decisions" made or stored representing the sub-problem. In translating propagation to Boolean reasoning these decisions become the Boolean variables. We introduce a, novel to our knowledge, encoding of integer domains as Booleans, combining the DIMACS encoding (see e.g. [18]) with that of [2]. It uses Boolean variables $[\![x = d]\!], d \in D_{init}(x)$, $[\![x \leqslant d]\!], \min D_{init}(x) \leqslant d < \max D_{init}(x)$. Set bounds domains are encoded as usual with the Boolean variables $[\![e \in S_i]\!], e \in \max D_{init}(S_i)$.

The Boolean variables directly represent changes to domains made by atomic constraints. Let *lit* be the mapping of atomic constraints to Boolean literals. We define

$$\begin{aligned} lit(false) &= \bot \\ lit(x_i = d) &= [\![x_i = d]\!] \\ lit(x_i \neq d) &= \neg[\![x_i = d]\!] \\ lit(e \in S_i) &= [\![e \in S_i]\!] \\ lit(e \notin S_i) &= \neg[\![e \in S_i]\!] \end{aligned}$$

$$lit(x_i \leqslant d) = \begin{cases} \top & d = \min D_{init}(x_i) \\ [\![x_i \leqslant d]\!] & \text{otherwise} \end{cases}$$

$$lit(x_i \geqslant d) = \begin{cases} \top & d = \max D_{init}(x_i) \\ \neg[\![x_i \leqslant d-1]\!] & \text{otherwise} \end{cases}$$

where $\min D_{init}(x_i) \leqslant d \leqslant \max D_{init}(x_i)$ 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 $l \neq \top$. Note also that for "Boolean" integers where $D_{init}(x) = [\,0\,..\,1\,]$ we have that $[\![x = 1]\!] \leftrightarrow \neg[\![x = 0]\!] \leftrightarrow \neg[\![x \leqslant 0]\!]$ so we can just use a single Boolean variable to represent the integer.

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:

$$assign(D, v) = \{lit(c) \mid v \in D(v) \models c, v \in vars(c)\}$$

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

5.2 Consistency of Domains

Representations of set variables are automatically consistent with respect to a set of literals, but this is not the case for representations of integer variables, since we could assert for example $[\![x = 3]\!]$ and $[\![x \leqslant 2]\!]$ simultaneously. For a variable x where $D_{init}(x) = [\,l\,..\,u\,]$ we maintain the consistency of assignment by adding the clauses $DOM(x)$:

$$\begin{aligned} \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d+1]\!] & \quad l \leqslant d < u-1 \\ \neg[\![x = d]\!] \vee [\![x \leqslant d]\!] & \quad l \leqslant d < u \\ \neg[\![x = d]\!] \vee \neg[\![x \leqslant d-1]\!] & \quad l < d \leqslant u \\ [\![x = l]\!] \vee \neg[\![x \leqslant l]\!] & \\ [\![x = d]\!] \vee \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d-1]\!] & \quad l < d < u \\ [\![x = u]\!] \vee [\![x \leqslant u-1]\!] & \end{aligned}$$

which encode $[\![x \leqslant d]\!] \rightarrow [\![x \leqslant d+1]\!]$ and $[\![x = d]\!] \leftrightarrow ([\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d-1]\!])$. For a set variable S, we define $DOM(S) = \{\}$, and then for all variables, $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

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

Theorem 1. *Let C be a set of atomic constraints on variable v, and $D = solv(\{true \rightarrowtail c | c \in C\}, D_{init})$ then $assign(D, v) = UP(\{\}, \{lit(c) \mid c \in C\} \cup DOM(v))$.*

The usual encoding of finite domains into SAT is the so called DIMACS encoding using only the variables $[\![x = d]\!]$ (see e.g. [18]). It enforces consistency of domains for $D_{init}(x) = [\, l \,..\, u \,]$ with the clause $\vee_{d=l}^{u} [\![x = d]\!]$ and the $O(n^2)$ clauses $\wedge_{l \leqslant d_1 < d_2 \leqslant u} \neg [\![x = d_1]\!] \vee \neg [\![x = d_2]\!]$. Note our encoding is linear and has equally strong unit propagation.

If for a variable x we are only interested in the atomic constraints $x \leqslant d$ and $x \geqslant d$ (i.e. bounds propagation on x) then we can omit the propositions $[\![x = d]\!]$ and the corresponding clauses from $DOM(x)$.

We can map unit propagation fixpoints of $DOM(v)$ to domains $D(v)$. Suppose $A = UP(A, DOM(v))$, then define $domain(A, v) = \{d \mid \forall l \in A.l$ involves $v, v = d \models l\}$.

We will be interested in minimal assignments that model a domain D. 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 4. 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. □

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 5. 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 2 requires a clause of ≈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 2. *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 $A = \overline{\bot}$ or $A \supseteq assign(D')$.*

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 $A = \overline{\bot}$ or $A \supseteq assign(solv(F, D))$.*

Example 6. 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 2. Then the clause C_0 defined in Example 5 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 2 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 approach 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 apriori representing a propagator f by a set of clauses, we execute the propagator during the SAT search and record what 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. This corresponds to a domain $D = domain(A)$. We then execute (individually) each propagator $f \in F$ on this domain obtaining 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.

We do not execute the propagation solver to fixpoint (although this is possible) because adding a single new clause may cause failure which means the work is wasted.

Given the above discussion we need to 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 $C \rightarrowtail c \in R$ not $D \models c$ (that is they generate new information). Ideally $R \subseteq rep(f)$ for some concise representation $rep(f)$ of the propagator f, but this may be difficult to achieve.

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.

Example 7. Consider the propagator f for $x_1 = x_2 \times x_3$ defined in Example 2. 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 8. Consider the propagator f for $x_0 \leftrightarrow x_1 + 1 \leqslant x_2$ from Example 3 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 3. 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 9. 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

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

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 $\mathit{lazy}(f)$ 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 $\mathit{lazy}(f)$ is usually straightforward. For example, Katsirelos and Bacchus [9] explain how to create $\mathit{lazy}(f)$ (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 [17], we can straightforwardly construct $\mathit{lazy}(f)$ since the indexical definition illustrates directly which atomic constraints contributed to the result. Direct constructions of $\mathit{lazy}(f)$ may not necessarily be tight. For propagators implemented using Binary Decision Diagrams we can automatically generate tight propagation rules using BDD operations [7]. 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 $\mathit{lazy}(f)$.

Example 10. We can make the propagation rules of Example 9 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

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

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 3. *Let* $A = UP(assign(D), DOM \cup \{cl(r) \mid r \in \cup_{f \in F} lazy(f)(D), \}$ *then* $A = \overline{\bot}$ *or* $A \supseteq assign(iter(F, D))$.

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 Building a Lazy Clause Generator System

We construct a lazy clause generator, by adding a cut-down propagation engine into a SAT solver. The interface between the propagators and the SAT solver is managed as follows. Each Boolean variable is associated with an integer or set atomic constraint. After the SAT solver reaches a fixpoint of unit propagation, we run over the newly fixed Boolean literals. For each Boolean literal l which is decided or inferred, we make the corresponding change to the domain defined by the atomic constraint $lit^{-1}(l)$. We queue the propagators possibly effected by the change, and then execute them. If we find a propagation that modifies the domain of some integer or set variable, we construct the propagation rule that explains it and add this as a clause permanently[4] to the SAT solver, and add its unit consequence to the SAT solvers literal queue (queue of decisions and unit consequences). If we find a clause that causes failure we immediately invoke the SAT solvers conflict resolution procedure. Otherwise when the queue is empty, we invoke the SAT solver on the new literals discovered by propagation, and the process repeats.

On failure for each Boolean literal l which is removed from the current assignment, we undo the change of atomic constraint $lit^{-1}(l)$. Note that since all individual domain changes are reflected in Boolean literals this is sufficient. For example 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.

8 Experiments

We have built a prototype lazy clause generator system using MiniSat [12] version 2.0 beta as the starting point. We give experiments using open-shop scheduling problems from [3]. The experiments are run on a 3GHz Intel Pentium D with 4Gb RAM running Debian Linux 3.1. 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

[4] We do not currently allow nogood minimzation to remove these clauses, though perhaps we should.

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

Benchmark	**Time(sec)**				**Conflict number**			**Clause ratio**	
	cut_sat	**csp2sat**	**sat**	**smt**	**cut_sat**	**csp2sat**	**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

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 $DOM(x)$. Using the full domain representation approximately doubles the computation time of the lazy approach. We use static translations for the first two kinds of constraints and lazy propagators for the reified difference inequalities.

We compare our lazy clause generation approach versus the static approach of [16] using MiniSat version 2.0 beta as the SAT solver, and versus the Barcelogic DPLL(T) solver version 1.1 using its difference logic theory solver [5]. We do not compare against other finite domain propagation solvers, because without very sophisticated encodings and search strategies [10], they are not competitive on these problems, since they lack nogoods.

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 [16] for details. We note that dichtomic optimization search is in a sense advantageous to the static approach since it generates clauses once which are effectively used in solving multiple (linked) satisfaction subproblems.

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 **cut_sat**, the static approach **csp2sat** (and just the time spend in the SAT solver for the static approach **sat**), and the SMT approach **smt**. We also give the number of conflicts for each approach, and the average and minimum across all subproblems in the dichotomic search of the ratio of clauses for the static 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 `csp2sat` 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 [10], so they are not considered easy for technologies without nogoods.

The open-shop scheduling suite `tai` shown in Table 2 is more difficult. As the problem size grows the advantage of the lazy and static approaches grows over

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

Benchmark	Time(sec)				Conflict number			Clause ratio	
	cut_sat	**csp2sat**	**sat**	**smt**	**cut_sat**	**csp2sat**	**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(sec)				Conflict number			Cl. ratio	
	cut_sat	**csp2sat**	**sat**	**smt**	**cut_sat**	**csp2sat**	**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

the SMT approach. The search space explored by the lazy approach is around twice that of the static 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 [16] are examined separately. The lazy approach is uniformly better than the static approach, and better than SMT on the larger problems (all `j7` and above except `j7-per10-1`). To save experimental time for the three hardest problems we only try to find a solution with optimal makespan (a single subproblem) (dichotomic search for the largest problem takes over 2 days for **csp2sat**). Surprisingly **csp2sat** improves on **cut_sat** 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.

Overall, **cut_sat** solves faster than **csp2sat** except for `j7-per0-0`, `j8-per10-2` and `j7-per10-2`. While it requires more search than **csp2sat**, the massive reduction in clauses pays off. The lowest clause ratio that occurs in any instance is 3.46. Overall **cut_sat** generally improves upon **smt** the harder the examples become.

Table 4. Linear equations examples (average of 100 runs)

Benchmark	**Time(sec)**		**Conflicts/Failures**	
	cut_sat	**gecode**	**cut_sat**	**gecode**
eq10	0.010	0.074	28	94
eq20	0.010	0.073	18	54
alpha	0.310	0.267	143	7435
money	0.004	0.069	29	3

Finally we also experimented with some well-known problems using (non-tight) bounds propagators for large linear equations (see Example 9). For none of these problems could the static approach generate the clauses within hours, and SMT modulo difference logic is not applicable, so we compare with Gecode 1.3.1 [6] a highly optimized propagation solver. The lazy approach uses an **alldifferent** propagator equivalent to bounds propagation on disequations ($x_1 \neq x_2$) and SAT VSIDS search, while Gecode uses its native `distinct` propagator and default labelling. Both solvers look for all solutions. The results are shown in Table 4. Clearly nogoods can substantially reduce the search for these problems, and the lazy approach is at least competitive with Gecode.

9 Related Work and Conclusion

The paper [16] explains how to statically 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 static 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.

The closest related work to this paper is the hybrid BDD and SAT bounds propagation set solver described in [7]. 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. [4], 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 [8] 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 [9] 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 [15] 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 [14] solver, where each propagator is considered as a separate theory, and theory propagation is used to learn clauses.

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. But we have only really scratched the surface of the possibilities of the lazy approach.

References

1. C.W. Choi, J.H.M. Lee, and P. J. Stuckey. Propagation redundancy in redundant modelling. In *Proceedings of CP-2003*, volume 2833 of *LNCS*, pages 229–243, 2003.
2. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Procs. AAAI-94*, pages 1092–1097, 1994.
3. CSP2SAT. `http://bach.istc.kobe-u.ac.jp/csp2sat/`. [Dec06].
4. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
5. Barcelogic for SMT. `www.lsi.upc.es/~oliveras/bclt-main.html`. [Feb07].
6. GECODE. `www.gecode.org`. [Feb07].
7. P. Hawkins and P.J. Stuckey. A hybrid BDD and SAT finite domain constraint solver. In *Proceedings PADL06*, volume 3819 of *LNCS*, pages 103–117, 2006.
8. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of CP2003*, volume 2833 of *LNCS*, pages 873–877, 2003.
9. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *The Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 390–396, 2005.
10. P. Laborie. Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings IJCAI 2005*, pages 181–186, 2005.
11. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
12. MiniSat. `www.cs.chalmers.se/Cs/Resarch/FormalMethods/MiniSat/`. [Dec06].
13. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC-2001*, 2001.
14. R. Niewenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *LPAR'04*, volume 3452 of *LNAI*, pages 36–50, 2004.
15. O. Roussel. 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*, pages 83–88. 2005.
16. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP to SAT. In *Proceedings of CP-2006*, volume 4204 of *LNCS*, pages 590–603, 2006.
17. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). *JLP*, 37(1–3):139–164, 1998.
18. T. Walsh. SAT v CSP. In *Proceedings of CP-2000*, volume 1894 of *LNCS*, pages 441–456, 2000.