Efficient Constraint Propagation Engines

CHRISTIAN SCHULTE
KTH - Royal Institute of Technology, Sweden
and
PETER J. STUCKEY
NICTA Victoria Laboratory, University of Melbourne, Australia

This paper presents a model and implementation techniques for speeding up constraint propagation. Three fundamental approaches to improving constraint propagation based on propagators as implementations of constraints are explored: keeping track of which propagators are at fixpoint, choosing which propagator to apply next, and how to combine several propagators for the same constraint.

We show how idempotence reasoning and events help track fixpoints more accurately. We improve these methods by using them dynamically (taking into account current variable domains to improve accuracy). We define priority-based approaches to choosing a next propagator and show that dynamic priorities can improve propagation. We illustrate that the use of multiple propagators for the same constraint can be advantageous with priorities, and introduce staged propagators that combine the effects of multiple propagators with priorities for greater efficiency.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Constructs and Features—*Constraints*; D.3.3 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Inference engines*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Scheduling*

General Terms: Languages, Design, Experimentation, Performance

Additional Key Words and Phrases: Constraint (logic) programming, constraint propagation, events, finite domain constraints, fixpoint reasoning, priorities

---

1. INTRODUCTION

We consider the problem of solving *Constraint Satisfaction Problems* (CSPs) defined in the sense of Mackworth [1977], which can be stated briefly as follows:

> We are given a set of variables, a domain of possible values for each vari-

---

able, and a set (read as a conjunction) of constraints. Each constraint
is a relation defined over a subset of the variables, limiting the combi-
nation of values that the variables in this subset can take. The goal is
to find a *consistent* assignment of values to the variables so that all the
constraints are satisfied simultaneously.

One widely-adopted approach to solving CSPs combines backtracking tree search
with constraint propagation. This framework is realized in finite domain constraint
programming systems, such as SICStus Prolog [Intelligent Systems Laboratory
2004], ILOG Solver [ILOG S.A. 2000], and Gecode [Gecode Team 2006] that have
been successfully applied to many real-life industrial applications.

At the core of a finite domain constraint programming system is a constraint
propagation engine that repeatedly executes propagators for the constraints of a
problem. Propagators discover and remove values from the domains of variables
that can no longer take part in a solution of the constraints.

Example 1.1. Consider a simple CSP, with variables x_1, x_2, and x_3 whose do-
main of possible values are respectively $x_1 \in \{2, 3, 4\}$, $x_2 \in \{0, 1, 2, 3\}$, $x_3 \in$
$\{-1, 0, 1, 2\}$ and the constraints are $x_3 = x_2$, $x_1 \le x_2 + 1$, and $x_1 \ne 3$.

A propagator for $x_3 = x_2$ can determine that $x_2 \ne 3$ in any solution of this
constraint since x_3 cannot take the value 3. Similarly, $x_3 \ne -1$. The propagator
then reduces the domains of the variables to $x_2 \in \{0, 1, 2\}$ and $x_3 \in \{0, 1, 2\}$. A
propagator for $x_1 \le x_2 + 1$ can determine that $x_1 \ne 4$ since $x_2 \le 2$, and $x_2 \ne 0$ since
$x_1 \ge 2$, so the domains are reduced to $x_1 \in \{2, 3\}$ and $x_2 \in \{1, 2\}$. The propagator
for $x_1 \ne 3$ can remove the value 3 from the domain of x_1, leaving $x_1 \in \{2\}$ (or
$x_1 = 2$). If we now reconsider the propagator for $x_3 = x_2$, we can reduce domains
to $x_2 \in \{1, 2\}$ and $x_3 \in \{1, 2\}$. No propagator can remove any further values.

We have not solved the problem, since we do not know a value for each variable.
So after propagation we apply search, usually by splitting the domain of a variable
into two disjoint subsets and considering the resulting two subproblems.

Suppose we split the domain of x_2. One subproblem has $x_1 \in \{2\}$, $x_2 \in \{1\}$,
and $x_3 \in \{1, 2\}$. Applying the propagator for $x_2 = x_3$ results in $x_3 \in \{1\}$. Since
each variable now takes a fixed value we can check that $x_1 = 2, x_2 = 1, x_3 = 1$ is a
solution to the CSP. The other subproblem has $x_1 \in \{2\}$, $x_2 \in \{2\}$ and $x_3 \in \{1, 2\}$,
and leads to another solution.   □

As can be seen from the example, finite domain constraint programming in-
terleaves propagation with search. In this paper we investigate how to make a
propagation engine as efficient as possible.

There are two important decisions the engine must make: which propagators
should execute, and in which order should they execute. To make constraint prop-
agation efficient, it is clear that the engine needs to take the following issues into
account: avoid unnecessary propagator execution, restrict propagation to relevant
variables, and choose the cheapest possible method for propagation. In this pa-
per we show how propagation can be sped up if the engine takes these issues into
account.

The contributions of the paper are as follows:

— We give a formal definition of propagation systems including fixpoint and

event-based optimizations used in current propagation systems.

— We extend event-based propagation systems to use dynamically changing event sets.

— We introduce multiple propagators and staged propagators for a single constraint for use with propagation queues with priority.

— We give experimental results that clarify the impact of many choices in implementing propagation engines: including idempotence reasoning, static and dynamic events, basic queuing strategies, priority queues, and staged propagation.

Plan of the paper. The next section introduces propagation-based constraint solving, followed by a model for constraint propagation systems in Section 3. Section 4 presents how to optimize propagation by taking idempotence into account, while Section 5 explores the use of event sets. Which propagator should be executed next is discussed in Section 6, while combination strategies of multiple propagators for the same constraint is discussed in Section 7. Experiments for each feature are included in the relevant section, and a summary is given in Section 8. Section 9 concludes.

2.   PROPAGATION-BASED CONSTRAINT SOLVING

This section defines our terminology for the basic components of a constraint propagation engine. In this paper, we restrict ourselves to finite domain integer constraint solving. Almost all the discussion applies to other forms of finite domain constraint solving such as for sets and multisets.

Domains. A *domain D* is a complete mapping from a fixed (finite) set of variables $\mathcal{V}$ to finite sets of integers. A *false domain D* is a domain with $D(x) = \emptyset$ for some $x \in \mathcal{V}$. A variable $x \in \mathcal{V}$ is *fixed* by a domain D, if $|D(x)| = 1$. The *intersection* of domains D_1 and D_2, denoted $D_1 \sqcap D_2$, is defined by the domain $D(x) = D_1(x) \cap D_2(x)$ for all $x \in \mathcal{V}$.

A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$. A domain D_1 is stronger than (equal to) a domain D_2 w.r.t. variables V, denoted $D_1 \sqsubseteq_V D_2$ (resp. $D_1 =_V D_2$), if $D_1(x) \subseteq D_2(x)$ (resp. $D_1(x) = D_2(x)$) for all $x \in V$.

A range is a contiguous set of integers, we use *range* notation $[l \mathinner{..} u]$ to denote the range $\{d \in \mathbb{Z} \mid l \leq d \leq u\}$ when l and u are integers. A domain is a *range domain* if $D(x)$ is a range for all x. Let $D' = \mathrm{range}(D)$ be the smallest range domain containing D, that is, the unique domain $D'(x) = [\inf D(x) \mathinner{..} \sup D(x)]$ for all $x \in \mathcal{V}$.

We shall be interested in the notion of an *initial domain*, which we denote D_{init}. The initial domain gives the initial values possible for each variable. It allows us to restrict attention to domains D such that $D \sqsubseteq D_{\mathrm{init}}$.

Valuations and constraints. An *integer valuation θ* is a mapping of variables to integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions and constraints involving the variables in the natural way.

Let vars be the function that returns the set of variables appearing in a valuation. We define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(x_i) \in D(x_i)$ for all $x_i \in \mathrm{vars}(\theta)$.

The *infimum* and *supremum* of an expression e with respect to a domain D are defined as $\inf_D e = \inf\{\theta(e) \mid \theta \in D\}$ and $\sup_D e = \sup\{\theta(e) \mid \theta \in D\}$.

We can map a valuation θ to a domain D_θ as follows

$$D_\theta(x) = \begin{cases} \{\theta(x)\} & x \in \mathrm{vars}(\theta) \\ D_{\mathrm{init}}(x) & \text{otherwise} \end{cases}$$

A *constraint c* over variables $x_1, \ldots, x_n$ is a set of valuations θ such that $\mathrm{vars}(\theta) = \{x_1, \ldots, x_n\}$. We also define $\mathrm{vars}(c) = \{x_1, \ldots, x_n\}$.

Propagators. We will *implement* a constraint c by a set of propagators $\mathrm{prop}(c)$ that map domains to domains. A *propagator f* is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. A propagator f is *correct* for a constraint c iff for all domains D

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

This is a very weak restriction, for example the identity propagator is correct for all constraints c.

A set of propagators F is *checking* for a constraint c, if for all valuations θ where $\mathrm{vars}(\theta) = \mathrm{vars}(c)$ the following holds: $f(D_\theta) = D_\theta$ for all $f \in F$, iff $\theta \in c$. That is, for any domain D_θ corresponding to a valuation on $\mathrm{vars}(c)$, $f(D_\theta)$ is a fixpoint iff θ is a solution of c. We assume that $\mathrm{prop}(c)$ is a set of propagators that is correct and checking for c.

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

Example 2.1 *Propagators, input, and output.* For the constraint $c \equiv x_1 \leq x_2 + 1$ the function f_a defined by $f_a(D)(x_1) = \{d \in D(x_1) \mid d \leq \sup_D x_2 + 1\}$ and $f_a(D)(x) = D(x), x \neq x_1$ is a correct propagator for c. Its output variables are $\{x_1\}$ and its input variables are $\{x_2\}$. Let $D_1(x_1) = \{1, 5, 8\}$ and $D_1(x_2) = \{1, 5\}$, then $f(D_1) = D_2$ where $D_2(x_1) = D_2(x_2) = \{1, 5\}$.

The propagator f_b defined as $f_b(D)(x_2) = \{d \in D(x_2) \mid d \geq \inf_D x_1 - 1\}$ and $f_b(D)(x) = D(x), x \neq x_2$ is another correct propagator for c. Its output variables are $\{x_2\}$ and input variables $\{x_1\}$.

The set $\{f_a, f_b\}$ is checking for c. The domain $D_{\theta_1}(x_1) = D_{\theta_1}(x_2) = \{2\}$ corresponding to the solution $\theta_1 = \{x_1 \mapsto 2, x_2 \mapsto 2\}$ of c is a fixpoint of both propagators. The non-solution domain $D_{\theta_2}(x_1) = \{2\}$, $D_{\theta_2}(x_2) = \{0\}$ corresponding to the valuation $\theta_2 = \{x_1 \mapsto 2, x_2 \mapsto 0\}$ is not a fixpoint (of either propagator). $\quad\square$

A *propagation solver* $\mathrm{solv}(F, D)$ for a set of propagators F and an initial domain D finds the greatest mutual fixpoint of all the propagators $f \in F$. In other words, $\mathrm{solv}(F, D)$ returns a new domain defined by

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

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

Note that by inverting the direction of $\sqsubseteq$ we could equally well phrase this as a least fix point (as in [Apt 2003]). But the current presentation emphasizes the *reduction* of domains as computation progresses.

Domain and bounds propagators. A consistency notion C gives a condition on domains with respect to constraints. A set of propagators F maintains C-*consistency* for a constraint c, if $\mathrm{solv}(F, D)$ is always C consistent for c. Many propagators in practice are designed to maintain some form of consistency: usually domain or bounds. But note that many more do not.

The most successful consistency technique is *arc consistency* [Mackworth 1977], which ensures that for each binary constraint, every value in the domain of the first variable, has a supporting value in the domain of the second variable that satisfied the constraint. Arc consistency can be naturally extended to constraints of more than two variables. This extension has been called *generalized arc consistency* [Mohr and Masini 1988], as well as *domain consistency* [Van Hentenryck et al. 1991; 1998] (which is the terminology we will use), and *hyper-arc consistency* [Marriott and Stuckey 1998]. A domain D is *domain consistent* for a constraint c if D is the least domain containing all solutions $\theta \in D$ of c, that is, there does not exist $D' \sqsubset D$ such that $\theta \in D \wedge \theta \in c \rightarrow \theta \in D'$.

Define the *domain propagator* $\mathrm{dom}(c)$, for a constraint c as

$$
\begin{aligned}
\mathrm{dom}(c)(D)(x) &= \{\theta(x) \mid \theta \in D \wedge \theta \in c\} &&\text{where } x \in \mathrm{vars}(c) \\
\mathrm{dom}(c)(D)(x) &= D(x) &&\text{otherwise}
\end{aligned}
$$

The basis of bounds consistency is to relax the consistency requirement to apply only to the lower and upper bounds of the domain of each variable x. There are a number of different notions of bounds consistency [Choi et al. 2006], we give the two most common here.

A domain D is bounds$(\mathbb{Z})$ *consistent* for a constraint c, $\mathrm{vars}(c) = \{x_1, \ldots, x_n\}$, if for each variable x_i, $1 \le i \le n$ and for each $d_i \in \{\inf_D x_i, \sup_D x_i\}$ there exist *integers* d_j with $\inf_D x_j \le d_j \le \sup_D x_j$, $1 \le j \le n, j \ne i$ such that $\theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is an *integer solution* of c.

A domain D is bounds$(\mathbb{R})$ *consistent* for a constraint c , $\mathrm{vars}(c) = \{x_1, \ldots, x_n\}$, if for each variable $x_i, 1 \le i \le n$ and for each $d_i \in \{\inf_D x_i, \sup_D x_i\}$ there exist *real numbers* d_j with $\inf_D x_j \le d_j \le \sup_D x_j$, $1 \le j \le n, j \ne i$ such that $\theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is a *real solution* of c.

A set of propagators F maintains bounds(α) *consistency* for a constraint c, if for all domains D, $\mathrm{solv}(F, D)$ is bounds(α) consistent for c.

We can define a bounds$(\mathbb{Z})$ *propagator*, $\mathrm{zbnd}(c)$ for a constraint c as follows:

$$
\mathrm{zbnd}(c)(D) = D \sqcap \mathrm{range}(\mathrm{dom}(c)(\mathrm{range}(D)))
$$

It is not straightforward to give a generic description of bounds$(\mathbb{R})$ propagators, $\mathrm{rbnd}(c)$, for a constraint c, that just maintains bounds$(\mathbb{R})$ consistency. Examples 3.2 and 4.3 define three such propagators.
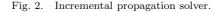
3. CONSTRAINT PROPAGATION SYSTEMS

A constraint propagation system evaluates the function $\mathrm{solv}(F, D)$ during backtracking search. We assume an execution model for solving a constraint problem

```
search(F_o, F_n, D)
    D := isolv(F_o, F_n, D)                                    % propagation
    if (D is a false domain)
        return false
    if (∃x ∈ V.|D(x)| > 1)
        choose {c_1, ..., c_m} where C ∧ D ⊨ c_1 ∨ ··· ∨ c_m    % search strategy
        for i ∈ [1 .. m]
            if (search(F_o ∪ F_n, prop(c_i), D))
                return true
        return false
    return true
```

<div align="center">Fig. 1. Search procedure</div>

```
isolv(F_o, F_n, D)
    F := F_o ∪ F_n; Q := F_n
    while (Q ≠ ∅)
        f := choose(Q)                    % select next propagator to apply
        Q := Q − {f}; D' := f(D)
        Q := Q ∪ new(f, F, D, D')         % add propagators f' ∈ F ...
        D := D'                           % ... not necessarily at fixpoint at D'
    return D
```

<div align="center">Fig. 2. Incremental propagation solver.</div>

with a set of constraints C and an initial domain D_0 as follows. We execute the procedure $\mathsf{search}(\emptyset, F, D_0)$ implementing depth-first search given in Figure 1 for an initial set of propagators $F = \cup_{c \in C} \mathrm{prop}(c)$. This procedure is used to make precise the optimizations presented in the remainder of the paper.

Note that the propagators are partitioned into two sets, the old propagators F_o and the new propagators F_n. The *incremental* propagation solver $\mathsf{isolv}(F_o, F_n, D)$ (to be presented later) takes advantage of the fact that D is guaranteed to be a fixpoint of the old propagators F_o.

The somewhat unusual definition of search is quite general. The default search strategy for many problems is to choose a variable x such that $|D(x)| > 1$ and explore $x = \inf_D x$ or $x \geq \inf_D x + 1$. This is commonly thought of as changing the domain D for x to either $\{\inf_D x\}$ or $\{d \in D(x) \mid d > \inf_D x\}$. This framework allows more general strategies, for example $x_1 \leq x_2$ or $x_1 > x_2$.

The basic incremental propagation solver algorithm is given in Figure 2. The algorithm uses a queue Q of propagators to apply. Initially, Q contains the new propagators. Each time the while loop is executed, a propagator f is deleted from the queue, f is applied, and then all propagators that may no longer be at a fixpoint at the new domain D' are added to the queue. An invariant of the algorithm is that at the while statement $f(D) = D$ for all $f \in F - Q$.

The propagation solver isolv leaves two components undefined: $\mathsf{choose}(Q)$ chooses the propagator $f \in Q$ to be applied next; $\mathsf{new}(f, F, D, D')$ determines the set of propagators $f' \in F$ that are not guaranteed to be at their fixpoint at the domain D'. The remainder of the paper investigates how to best implement these two components.

Table I. Incremental propagation trace.

Q	f	$D'(x_1)$	$D'(x_2)$	$D'(x_3)$	$\mathsf{new}_{\mathrm{input}}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_c	$[0 .. 17]$	$\mathbf{[0 .. 8]}$	$[0 .. 6]$	$\{f_\mathrm{c}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_d	$[0 .. 17]$	$[0 .. 8]$	$\mathbf{[0 .. 5]}$	$\{f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_c	$\mathbf{[0 .. 16]}$	$[0 .. 8]$	$[0 .. 5]$	$\{f_\mathrm{c}, f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_d	$\mathbf{[0 .. 15]}$	$[0 .. 8]$	$[0 .. 5]$	$\{f_\mathrm{c}, f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_c	$[0 .. 15]$	$\mathbf{[0 .. 7]}$	$[0 .. 5]$	$\{f_\mathrm{c}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_d	$[0 .. 15]$	$[0 .. 7]$	$[0 .. 5]$	$\{\}$
$\{f_\mathrm{c}\}$	f_c	$\mathbf{[0 .. 14]}$	$[0 .. 7]$	$[0 .. 5]$	$\{f_\mathrm{c}, f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_d	$[0 .. 14]$	$[0 .. 7]$	$\mathbf{[0 .. 4]}$	$\{f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_c	$[0 .. 14]$	$[0 .. 7]$	$[0 .. 4]$	$\{\}$
$\{f_\mathrm{d}\}$	f_d	$\mathbf{[0 .. 12]}$	$[0 .. 7]$	$[0 .. 4]$	$\{f_\mathrm{c}, f_\mathrm{d}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_c	$[0 .. 12]$	$\mathbf{[0 .. 6]}$	$[0 .. 4]$	$\{f_\mathrm{c}\}$
$\{f_\mathrm{c}, f_\mathrm{d}\}$	f_d	$[0 .. 12]$	$[0 .. 6]$	$[0 .. 4]$	$\{\}$
$\{f_\mathrm{c}\}$	f_c	$[0 .. 12]$	$[0 .. 6]$	$[0 .. 4]$	$\{\}$

3.1 Basic Variable Directed Propagation

The core aim of the constraint propagation solver $\mathsf{solv}(F, D)$ is to find a domain that is a mutual fixpoint of all $f \in F$. The incremental solver $\mathsf{isolv}(F_o, F_n, D)$ already takes into account that initially D is a fixpoint of propagators $f \in F_o$. The role of new is (generally) to return *as few* propagators $f \in F$ as possible.

A basic definition of new is as follows

$$\mathsf{new}_{\mathrm{input}}(f, F, D, D') = \{f' \in F \mid \mathrm{input}(f') \cap \{x \in \mathcal{V} \mid D(x) \neq D'(x)\} \neq \emptyset\}$$

Here all propagators f' are added whose input variable domains have changed. By the definition of input variables, if none of them have changed for f', then $f'(D') = D'$ since $f'(D) = D$ if $f' \in F - Q$.

PROPOSITION 3.1. $\mathsf{new}_{\mathrm{input}}$ *maintains the invariant* $f'(D) = D$ *for all* $f' \in F - Q$ *at the start of the* **while** *loop.*

PROOF. Consider $f' \in F - Q$. Then $f'(D) = D$ and if $D =_{\mathrm{input}(f')} D'$ we have that $D' \sqcap f'(D) =_{\mathrm{output}(f')} f'(D') \sqcap D$. Then

$$
\begin{aligned}
D' &= & D' \sqcap D & \quad \text{since } D' \sqsubseteq D \\
&= & D' \sqcap f'(D) & \quad \text{since } D = f'(D) \\
&=_{\mathrm{output}(f')} & f'(D') \sqcap D & \quad \text{by definition of input}(f') \\
&= & f'(D') & \quad \text{since } f'(D') \sqsubseteq D' \sqsubseteq D
\end{aligned}
$$

Now $D' =_{\mathrm{output}(f')} f'(D')$ implies $D' = f'(D')$ by the definition of output(f'). Hence each f' in $F - Q$ is at fixpoint at the start of the loop. □

The incremental propagation solver isolv with this definition of new (assuming $F_o = \emptyset$) is more or less equivalent to the propagation algorithms in [Benhamou 1996] and [Apt 2003, page 267].

Example 3.2 *Incremental propagation.* Consider the problem with constraints

$c_c \equiv x_1 = 2x_2$ and $c_d \equiv x_1 = 3x_3$ represented by the bounds($\mathbb{R}$) propagators

$$
\begin{aligned}
f_c(D)(x_1) &= D(x_1) \cap [2 \inf_D x_2 \,..\, 2 \sup_D x_2], \\
f_c(D)(x_2) &= D(x_2) \cap \left[\left\lceil \tfrac{1}{2} \inf_D x_1 \right\rceil \,..\, \left\lfloor \tfrac{1}{2} \sup_D x_1 \right\rfloor\right], \\
f_c(D)(x) &= D(x) && x \notin \{x_1, x_2\}
\end{aligned}
$$

$$
\begin{aligned}
f_d(D)(x_1) &= D(x_1) \cap [3 \inf_D x_3 \,..\, 3 \sup_D x_3], \\
f_d(D)(x_3) &= D(x_3) \cap \left[\left\lceil \tfrac{1}{3} \inf_D x_1 \right\rceil \,..\, \left\lfloor \tfrac{1}{3} \sup_D x_1 \right\rfloor\right], \\
f_d(D)(x) &= D(x) && x \notin \{x_1, x_3\},
\end{aligned}
$$

with initial domain $D(x_1) = [0 \,..\, 17]$, $D(x_2) = [0 \,..\, 9]$, and $D(x_3) = [0 \,..\, 6]$.

Initially, no propagator is known to be at fixpoint, so $Q = \{f_c, f_d\}$. Table I shows how incremental propagation proceeds. Column Q shows the value of Q just before the operation choose(Q) and f, D', and new$_{\text{input}}$ (for new$_{\text{input}}(f, F, D, D')$) show the values at each iteration of the **while** loop. □

4. FIXPOINT REASONING

The propagation engine computes a mutual fixpoint of all the propagators. Clearly, if we can determine that some propagators are at fixpoint without executing them we can limit the amount of work required by the engine.

4.1 Static Fixpoint Reasoning

A propagator f is *idempotent* if $f(D) = f(f(D))$ for all domains D. That is, applying f to any domain D yields a fixpoint of f.

Example 4.1 *Idempotent propagator*. The propagator f_e defined by

$$
\begin{aligned}
f_e(D)(x_1) &= \{d \in D(x_1) \mid \tfrac{3}{2}d \in D(x_2)\} \\
f_e(D)(x_2) &= \{d \in D(x_2) \mid \tfrac{2}{3}d \in D(x_1)\} \\
f_e(D)(x) &= D(x) && x \notin \{x_1, x_2\}
\end{aligned}
$$

is the domain propagator for the constraint $3x_1 = 2x_2$. The propagator f_e is idempotent. □

It is not difficult to see that each domain propagator dom(c) is idempotent.

PROPOSITION 4.2. *For all constraints c and domains D*

$$\text{dom}(c)(D) = \text{dom}(c)(\text{dom}(c)(D))$$

PROOF. Consider $\theta \in c$ where $\theta \in D$. Then by definition $\theta(x) \in \text{dom}(c)(D)$ for all $x \in \text{vars}(c)$. Hence $\theta \in \text{dom}(c)(D)$. Since $\text{dom}(c)(D) \sqsubseteq D$ we have $\theta \in c \wedge \theta \in D$ iff $\theta \in c \wedge \theta \in \text{dom}(c)(D)$. Hence $\text{dom}(c)(D) = \text{dom}(c)(\text{dom}(c)(D))$. □

Example 4.3 *Non-idempotent propagators*. While many propagators are idempotent, some widely used ones are *not* idempotent. Consider the constraint $3x_1 = 2x_2$ and the propagator f_f:

$$
\begin{aligned}
f_f(D)(x_1) &= D(x_1) \cap \left[\left\lceil \tfrac{2}{3} \inf_D x_2 \right\rceil \,..\, \left\lfloor \tfrac{2}{3} \sup_D x_2 \right\rfloor\right] \\
f_f(D)(x_2) &= D(x_2) \cap \left[\left\lceil \tfrac{3}{2} \inf_D x_1 \right\rceil \,..\, \left\lfloor \tfrac{3}{2} \sup_D x_1 \right\rfloor\right] \\
f_f(D)(x) &= D(x) && x \notin \{x_1, x_2\}
\end{aligned}
$$

Table II.   Incremental propagation trace with idempotent propagators.

Q	f	$D'(x_1)$	$D'(x_2)$	$D'(x_3)$	$\mathsf{new_{sfix}}$
$\{f_c^*, f_d^*\}$	f_c^*	[**0 .. 16**]	[**0 .. 8**]	[0 .. 6]	$\{\}$
$\{f_d^*\}$	f_d^*	[**0 .. 15**]	[0 .. 8]	[**0 .. 5**]	$\{f_c^*\}$
$\{f_c^*\}$	f_c^*	[**0 .. 14**]	[**0 .. 7**]	[0 .. 5]	$\{f_d^*\}$
$\{f_d^*\}$	f_d^*	[**0 .. 12**]	[0 .. 7]	[**0 .. 4**]	$\{f_c^*\}$
$\{f_c^*\}$	f_c^*	[0 .. 12]	[**0 .. 6**]	[0 .. 4]	$\{\}$

In almost all constraint programming systems $\mathrm{prop}(3x_1 = 2x_2)$ is $\{f_f\}$ where f_f is the bounds($\mathbb{R}$) propagator for $3x_1 = 2x_2$. Now f_f is not idempotent. Consider $D(x_1) = [0 .. 3]$ and $D(x_2) = [0 .. 5]$. Then $D' = f_f(D)$ is defined by $D'(x_1) = [0 .. 3] \cap [0 .. \lfloor 10/3 \rfloor] = [0 .. 3]$ and $D'(x_2) = [0 .. 5] \cap [0 .. \lfloor 9/2 \rfloor] = [0 .. 4]$. Now $D'' = f_f(D')$ is defined by $D''(x_1) = [0 .. 3] \cap [0 .. \lfloor 8/3 \rfloor] = [0 .. 2]$ and $D''(x_2) = [0 .. 4] \cap [0 .. \lfloor 9/2 \rfloor] = [0 .. 4]$. Hence $f_f(f_f(D)) = D'' \neq D' = f_f(D)$.   □

We can always create an idempotent propagator f' from a propagator f by defining $f'(D) = \mathrm{solv}(\{f\}, D)$. Indeed, in some implementations (for example [Harvey and Stuckey 2003]) $\mathrm{prop}(3x_1 = 2x_2)$ is defined as the fixpoint of applying f_f.

Assume that $\mathrm{idem}(f) = \{f\}$ if f is an idempotent propagator and $\mathrm{idem}(f) = \emptyset$ otherwise. The definition of new is improved by taking idempotence into account

$$\mathsf{new_{sfix}}(f, F, D, D') = \mathsf{new_{input}}(f, F, D, D') - \mathrm{idem}(f)$$

An idempotent propagator is never put into the queue after application.

Note that without the idempotence optimization each propagator f that changes the domain is likely to be executed again to check it is at fixpoint. Almost all constraint propagation solvers take into account static fixpoint reasoning (for example ILOG Solver [ILOG S.A. 2000], Choco [Laburthe 2000], SICStus [Intelligent Systems Laboratory 2004], and Gecode [Gecode Team 2006]). Some systems even only allow idempotent propagators (for example Mozart [Mozart Consortium 1999]).

Example 4.4 *Idempotent incremental propagation.* Consider the problem from Example 3.2 where idempotent propagators $f_c^*(D) = \mathrm{solv}(\{f_c\}, D)$ and $f_d^*(D) = \mathrm{solv}(\{f_d\}, D)$ are used.

Starting from the same domain $D(x_1) = [0 .. 17]$, $D(x_2) = [0 .. 9]$, and $D(x_3) = [0 .. 6]$ incremental propagation using $\mathsf{new_{sfix}}$ is shown in Table II.   □

4.2   Dynamic Fixpoint Reasoning

Even if a propagator is not idempotent we can often determine that $f(D)$ is a fixpoint of f for a specific domain D. For simplicity, we assume a function $\mathrm{fix}(f, D)$ that returns $\{f\}$ if it can show that $f(D)$ is a fixpoint for f and $\emptyset$ otherwise (of course without calculating $f(f(D))$, otherwise we gain nothing). In practice, $\mathrm{fix}(f, D)$ will be included in the implementation of f.

$$\mathsf{new_{dfix}}(f, F, D, D') = \mathsf{new_{input}}(f, F, D, D') - \mathrm{fix}(f, D)$$

Example 4.5 *Dynamic idempotence.* Bounds propagation for linear equations on range domains guarantees that the propagator is at a fixpoint if there is no rounding required in determining new endpoints [Harvey and Stuckey 2003, Theorem 8].

We can define $\mathrm{fix}(f_\mathrm{f}, D)$ for the bounds propagator f_f from Example 4.3 for the constraint $3x_1 = 2x_2$, as returning $\{f_\mathrm{f}\}$ if any new bound does not require rounding, e.g. $2\inf_D x_2/3 = \lceil 2\inf_D x_2/3\rceil$ or $2\inf_D x_2/3 \le \inf_D x_1$ and similarly for the other three bounds.

Consider applying f_f to the domain D'' from the same example. Now $D''' = f_\mathrm{f}(D'')$ is defined by $D'''(x_1) = [0 .. 2] \cap [0 .. \lfloor 8/3\rfloor] = [0 .. 2]$ and $D'''(x_2) = [0 .. 4] \cap [0 .. \lfloor 6/2\rfloor] = [0 .. 3]$. Notice that the new bound $x_2 \le 3$ is obtained without rounding $\lfloor 6/2\rfloor = \lfloor 3\rfloor = 3$. In this case we are guaranteed that the propagator is at a fixpoint.   □

Note that the dynamic case extends the static case since for idempotent f it holds that $\mathrm{fix}(f, D) = \{f\}$ for all domains D. The dynamic fixpoint reasoning extensions are obviously correct, given Proposition 3.1.

PROPOSITION 4.6. $\mathrm{new}_\mathrm{dfix}$ *maintains the invariant $f(D) = D$ for all $f \in F - Q$ at the start of the* **while** *loop.*

A complexity of either form of fixpoint reasoning is that a great deal of care has to be taken when we claim a propagator is at fixpoint. This is particularly true for bounds propagators and for propagators computing with multiple occurrences of the same variable.

Example 4.7 *Falling into domain holes.* Consider a bounds propagator for $x_1 = x_2 + 1$ defined as

$$
\begin{aligned}
f_\mathrm{g}(D)(x_1) &= D(x_1) \cap [\inf_D x_2 + 1 .. \sup_D x_2 + 1] \\
f_\mathrm{g}(D)(x_2) &= D(x_2) \cap [\inf_D x_1 - 1 .. \sup_D x_1 - 1] \\
f_\mathrm{g}(D)(x) &= D(x) \qquad\qquad\qquad\qquad x \notin \{x_1, x_2\}
\end{aligned}
$$

We would expect this propagator to be idempotent since there is no rounding required. Consider the application of f_g to the domain $D(x_1) = \{0, 4, 5, 6\}$ and $D(x_2) = \{2, 3, 4, 5\}$. Then $f_\mathrm{g}(D) = D'$ where $D'(x_1) = \{4, 5, 6\}$ and $D'(x_2) = \{2, 3, 4, 5\}$. This is not a fixpoint for f_g because of the hole (that is $1, 2, 4 \notin D(x_1)$) in the original domain of x_1.   □

Example 4.8 *Multiple variable occurrences.* The `regular` constraint introduced in [Pesant 2004] constrains a sequence of variables to take values described by a regular expression (or a corresponding finite automaton). A common case for the `regular` constraint is to express cyclic patterns by performing propagation on a sequence of variables where some variables appear multiply.

Assume a propagator f_h propagating that the sequence of variables $\langle x_1, x_2, x_3\rangle$ conforms to the `regular` expression $(11|00)0$ (that is, the values of three variables form either the string 110 or 000).

For a domain D with $D(x_1) = D(x_2) = D(x_3) = \{0, 1\}$ propagation for the sequence $\langle x_1, x_2, x_3\rangle$ is obtained by checking which values are still possible for each variable by traversing the sequence. In this particular case, for $D' = f_\mathrm{h}(D)$ we have that $D'(x_3) = \{0\}$ and $D'(x_1) = D'(x_2) = \{0, 1\}$ and D' is a fixpoint for f_h.

Now assume a sequence $\langle x_1, x_2, x_1\rangle$ where x_1 appears twice. Using the same strategy as above, a single forward traversal of the sequence yields: $D' = f_\mathrm{h}(D)$ where $D'(x_1) = \{0\}$ and $D'(x_2) = \{0, 1\}$. However, D' is *not* a fixpoint for f_h.   □

Table III.   Fixpoint reasoning experiments.

Example	none		static		dynamic	
	time (ms)	steps	time	steps	time	steps
`all-interval-500`	118.31	503 036	−3.1%	−24.8%	−38.9%	−24.9%
`alpha`	106.56	272 398	−2.0%	−5.7%	−1.5%	−5.7%
`bibd-7-3-60`	2 279.36	1 335 657	−0.2%	−6.5%	−0.2%	−6.5%
`cars`	4.64	15 641	−0.4%	±0.0%	+0.1%	±0.0%
`crowded-chess-7`	624.25	806 664	−0.1%	±0.0%	−0.2%	±0.0%
`donald-b`	0.70	546	−2.5%	−8.4%	−10.4%	−16.5%
`donald-d`	30.37	46	−6.3%	−6.5%	−6.4%	−13.0%
`donald-v`	0.38	546	−5.9%	−16.5%	−5.1%	−16.5%
`golomb-10-b`	1 347.48	2 642 464	+4.7%	−17.9%	+0.1%	−17.5%
`golomb-10-d`	2 430.00	2 642 962	+4.5%	−18.0%	+4.3%	−18.0%
`graph-color`	35.87	9 344	−0.1%	±0.0%	−7.2%	−6.2%
`grocery`	55.41	2 299	+2.6%	−3.8%	+2.6%	−3.8%
`knights-10`	7.46	48 038	+0.9%	+2.2%	+0.9%	+2.2%
`minsort-200`	342.48	240 991	−11.3%	−8.3%	−5.1%	−16.9%
`o-latin-7-d`	574.36	387 060	±0.0%	−0.5%	−8.7%	−18.2%
`partition-32`	8 571.24	16 785 128	−3.7%	−18.8%	−9.2%	−19.7%
`photo`	108.87	422 206	−0.3%	−0.8%	−14.6%	−3.6%
`picture`	1 553.42	150 165	−4.1%	−13.1%	−3.4%	−20.5%
`queens-400`	4 433.12	31 424 152	+0.2%	±0.0%	±0.0%	±0.0%
`queens-400-a`	16.15	2 469	−1.2%	−16.2%	−1.2%	−16.2%
`sequence-500`	517.96	151 609	−0.1%	−0.1%	−0.1%	−0.1%
`square-5-d`	33 391.24	1 762 492	−6.2%	−16.0%	−6.1%	−16.1%
`square-7-b`	10 166.24	7 731 557	+2.4%	−14.5%	−6.8%	−14.7%
`square-7-v`	5 690.00	13 956 982	−4.1%	−16.9%	−4.3%	−16.9%
`warehouse`	0.74	2 486	−2.5%	−1.8%	−7.9%	−9.1%
average (all)	—	—	−1.6%	−8.9%	−5.6%	−11.5%

The two examples above describe common cases where a propagator is not idempotent but nevertheless computes a fixpoint in many cases. In these cases, dynamic fixpoint reasoning is beneficial. Static fixpoint reasoning would force these propagators to be never considered to be at fixpoint. With dynamic fixpoint reasoning the propagator can decide whether it is at fixpoint for a given domain or not.

In practice, dynamic fixpoint reasoning completely subsumes static fixpoint reasoning and is easy to implement: a propagator is extended to not only return the new domain but also a flag indicating whether the propagator has computed a fixpoint.

4.3   Fixpoint Reasoning Experiments

Table III shows runtime (walltime) and the number of propagation steps for the three different variants of fixpoint reasoning considered. The first column presents absolute runtime values in milliseconds and the number of propagation steps when no fixpoint reasoning is considered. The two remaining columns "static" and "dynamic" show the relative change to runtime and propagation steps when using static and dynamic fixpoint reasoning. For example, a relative change of +50% means that the system takes 50% more time or propagation steps, whereas a value of −50% means that the system takes only half the time or propagation steps. The

row "average (all)" gives the average (geometric mean) of the relative values for all examples. More information on the examples can be found in Appendix A and on the used platform in Appendix B.

Note that both static and dynamic fixpoint reasoning do not change the memory requirements for any of the examples.

Static fixpoint reasoning. While static fixpoint reasoning reduces the number of propagator executions by 8.9% in average, the reduction in runtime is modest by 1.6% in average. The reason that the reduction in propagation steps does not directly translate to a similar reduction in runtime is that the avoided steps tend to be cheap: after all, these are steps not performing any propagation as the propagator is already at fixpoint.

Note that the ratio between reduction in steps and reduction in runtime ultimately depends on the underlying system. In Gecode, the system used, the actual overhead for executing a propagator is rather low. In systems with higher overhead one can expect that the gain in runtime will be more pronounced.

Examples with significant reduction in runtime (such as `donald-d`, `minsort-200`, `picture`, and `square-5-d`) profit because the execution of costly propagators is avoided (domain consistent linear equations for `donald-d` and `square-5-d`; `regular` propagators for `picture`; minimum propagators involving up to 200 variables for `minsort`). The behavior of `golomb-10-b` and `golomb-10-d` is explained further below.

Dynamic fixpoint reasoning. As expected, both runtime as well as propagation steps are considerably smaller for dynamic fixpoint reasoning compared to static reasoning. This is in particular true for examples `donald-b` and `square-7-b` where a bounds-consistent `alldifferent` constraint can take advantage of reporting whether propagation has computed a fixpoint due to no domain holes as discussed in Example 4.7 (`all-interval-500` shows the same behavior due to the absolute value propagator used).

Influence of propagation order. Some examples show a considerable increase in runtime (in particular, `golomb-10-b` and `golomb-10-d`). This is due to the fact that the order in which propagators are executed changes: costly propagators are executed often while cheap propagators are executed less often (witnessed by the decrease in propagation steps).

Section 6 presents priorities that order execution according to propagator priorities. For now it is sufficient to note that when these priority inversion problems are avoided through the use of priorities, there is no increase in runtime. Table IV provides evidence for this. The table shows relative runtime and propagation steps (relative to no fixpoint reasoning as in Table III). When avoiding priority inversion it becomes clear that both static as well as dynamic fixpoint reasoning consistently improve the number of propagation steps and also runtime. The only exception is `picture` where even with priorities the considerable reduction in execution steps does not translate into a reduction in runtime. This is possibly due to a change in propagation order that affects runtime (that this can have a remarkable effect even with priorities is demonstrated in Section 6).

Table IV.   Fixpoint reasoning experiments with propagator priorities.

Example	static		dynamic	
	time	steps	time	steps
`all-interval-500`	−2.9%	−24.9%	−37.7%	−25.0%
`alpha`	−5.4%	−15.6%	−4.6%	−15.6%
`bibd-7-3-60`	−0.7%	−6.5%	−0.7%	−6.5%
`cars`	−0.2%	±0.0%	+0.1%	±0.0%
`crowded-chess-7`	−0.7%	±0.0%	−0.5%	±0.0%
`donald-b`	−4.8%	−22.9%	−11.8%	−29.4%
`donald-d`	−6.7%	−5.1%	−6.7%	−28.8%
`donald-v`	−5.6%	−16.5%	−5.3%	−16.5%
`golomb-10-b`	−4.9%	−23.5%	−6.5%	−23.5%
`golomb-10-d`	−3.3%	−23.4%	−3.0%	−23.4%
`graph-color`	+0.4%	±0.0%	−7.3%	−4.9%
`grocery`	−4.9%	−29.5%	−4.9%	−29.5%
`knights-10`	±0.0%	+1.3%	+0.2%	+1.3%
`minsort-200`	−10.9%	−9.1%	−0.9%	−9.2%
`o-latin-7-d`	+0.3%	−1.4%	−6.1%	−22.9%
`partition-32`	−8.5%	−30.4%	−11.8%	−30.9%
`photo`	−0.7%	−1.2%	−7.5%	−2.6%
`picture`	+6.5%	−13.1%	+4.0%	−20.5%
`queens-400`	+0.2%	±0.0%	+0.2%	±0.0%
`queens-400-a`	−1.3%	−16.2%	−1.3%	−16.2%
`sequence-500`	+0.1%	±0.0%	+0.1%	±0.0%
`square-5-d`	−6.4%	−11.1%	−7.3%	−15.8%
`square-7-b`	−2.5%	−25.2%	−10.0%	−26.0%
`square-7-v`	−6.6%	−24.9%	−5.8%	−24.9%
`warehouse`	−2.2%	−1.4%	−7.9%	−9.8%
average (all)	−2.9%	−12.7%	−6.1%	−15.9%

5.   EVENT REASONING

The next improvement is to consider what changes in domains of input variables can cause the propagator to no longer be at a fixpoint. To this end we use events: an *event* is a change in the domain of a variable.

Assume that the domain D changes to the domain $D' \sqsubseteq D$. A typical set of events defined in a constraint propagation system are:

— $\text{fix}(x)$: the variable x becomes fixed (or the domain D' becomes failed), that is $|D'(x)| \leq 1$ and $|D(x)| > 1$.

— $\text{lbc}(x)$: the lower bound of variable x changes, that is $\inf_{D'} x > \inf_D x$.

— $\text{ubc}(x)$: the upper bound of variable x changes, that is $\sup_{D'} x < \sup_D x$.

— $\text{dmc}(x)$: the domain of variable x changes, that is $D'(x) \subset D(x)$.

Clearly the events overlap. Whenever a $\text{fix}(x)$ event occurs then a $\text{lbc}(x)$ event, a $\text{ubc}(x)$ event, or both events must also occur. If any of the first three events occur then a $\text{dmc}(x)$ event occurs. These events satisfy the following property.

Definition 5.1 *Event.* An *event* ϕ is a change in domain defined by an event condition $\phi(D, D')$ which states that event ϕ occurs when the domain changes

from D to $D' \sqsubseteq D$. The event condition must satisfy the following property

$$\phi(D, D'') = \phi(D, D') \vee \phi(D', D'')$$

where $D'' \sqsubseteq D' \sqsubseteq D$. So an event occurs on a change from D to D'' iff it occurs in the change from D to D' or from D' to D''.

Given a domain D and a stronger domain $D' \sqsubseteq D$, then events(D, D') is the set of events ϕ where $\phi(D, D')$. Suppose $D'' \sqsubseteq D' \sqsubseteq D$, then clearly

$$\text{events}(D, D'') = \text{events}(D, D') \cup \text{events}(D', D''). \tag{1}$$

Most integer propagation solvers use the events defined above, although many systems collapse ubc(x) and lbc(x) into a single event bc(x) (for example, SIC-Stus [Intelligent Systems Laboratory 2004], ILOG Solver [ILOG S.A. 2000], and Gecode [Gecode Team 2006]). Choco [Laburthe 2000] maintains an event queue and interleaves propagator execution with events causing more propagators to be added to the queue.

Other kinds of events or variants of the above events are also possible. For example, (for domains D and D' with $D' \sqsubseteq D$):

— bc(x): as discussed above (lbc$(x) \vee$ ubc(x)).

— two(x): the variable x reduces to a domain of at most two values: $|D'(x)| \leq 2$ and $|D(x)| > 2$.

— ran(x): the variable x reduces to a domain that will always be a range (two consecutive values or a single value): $\sup_{D'} x - \inf_{D'} x \leq 1$ and $\sup_D x - \inf_D x > 1$.

— pos(x): the variable x reduces to a domain that is strictly positive, that is $\inf'_D x > 0$ and $\inf_D x \leq 0$ (likewise, an event neg(x) for reduction to a strictly negative domain).

— nneg(x): the variable x reduces to a domain that is non-negative: $\inf'_D x \geq 0$ and $\inf_D x < 0$ (likewise, an event npos(x) for reduction to a non-positive domain).

— neq(x, d): the variable x can no longer take the value d, that is $d \in D(x)$ and $d \notin D'(x)$

The events two and ran are useful for tracking endpoint-relevance and range-equivalence [Schulte and Stuckey 2005]. The neq event has been used in e.g. Choco [Laburthe 2000] and B-Prolog [Zhou 2006] for building AC4 [Mohr and Henderson 1986] style propagators.

Example 5.2 *Events.* Let $D(x_1) = \{1, 2, 3\}$, $D(x_2) = \{3, 4, 5, 6\}$, $D(x_3) = \{0, 1\}$, and $D(x_4) = \{7, 8, 10\}$ while $D'(x_1) = \{1, 2\}$, $D'(x_2) = \{3, 5, 6\}$, $D'(x_3) = \{1\}$ and $D'(x_4) = \{7, 8, 10\}$. Then events(D, D') is

$$\{\text{ubc}(x_1), \text{dmc}(x_1), \text{dmc}(x_2), \text{fix}(x_3), \text{lbc}(x_3), \text{dmc}(x_3)\}$$

Considering the additional events we obtain in addition

$$\{\text{bc}(x_1), \text{two}(x_1), \text{ran}(x_1), \text{bc}(x_3), \text{neq}(x_1, 3), \text{neq}(x_2, 4), \text{neq}(x_3, 0)\}$$

<div align="right">□</div>

Example 5.3 *Events are monotonic.* Events are *monotonic*: further changes to a domain do not discard events from previous changes. Consider the property

range(x) capturing that $D(x)$ is a range for a domain D (this property is related to the event ran(x), lacking the restriction that the domain can have at most two elements).

The property range(x) is *not* an event: consider domains $D'' \sqsubseteq D' \sqsubseteq D$ with $D(x) = \{1, 2, 3, 5\}$, $D'(x) = \{1, 2, 3\}$, and $D''(x) = \{1, 3\}$. If range were an event, then events$(D, D'') = $ events$(D, D') \cup$ events(D', D''). However,

$$\text{events}(D, D'') = \{\text{dmc}(x), \text{ubc}(x)\}$$

whereas

$$\text{events}(D, D') \cup \text{events}(D', D'') = \{\text{dmc}(x), \text{ubc}(x), \text{range}(x)\} \cup \{\text{dmc}(x)\}$$

This does not preclude a propagator to take advantage of the fact that $D(x)$ for a variable x is a range. The point is that the propagator cannot synchronize its execution on $D(x)$ becoming a range. □

5.1 Static Event Sets

Re-execution of certain propagators can be avoided since they require certain events to generate new information.

Definition 5.4 *Propagator dependence.* A propagator f is *dependent on* a set of events es(f) iff

(a) for all domains D: if $f(D) \neq f(f(D))$ then events$(D, f(D)) \cap$ es$(f) \neq \emptyset$,
(b) for all domains D and D' where $f(D) = D$ and $D' \sqsubseteq D$: if $f(D') \neq D'$ then events$(D, D') \cap$ es$(f) \neq \emptyset$.

The definition captures the following. If f is not at a fixpoint then one of the events in its event set occurs. If f is at a fixpoint D then any change to a domain that is not a fixpoint D' involves an occurrence of one of the events in its set. Note that for idempotent propagators the case (a) never occurs.

For convenience later we will store the event set chosen for a propagator f in an array evset[f].

Clearly, if we keep track of the events since the last invocation of a propagator, we do not need to apply a propagator if it is not dependent on any of these events.

Example 5.5 *Event sets.* Event sets for previously discussed propagators are as follows:

$$
\begin{array}{ll}
f_\text{a} & \{\text{ubc}(x_2)\} \\
f_\text{b} & \{\text{lbc}(x_1)\} \\
f_\text{e} & \{\text{dmc}(x_1), \text{dmc}(x_2)\} \\
f_\text{f} & \{\text{lbc}(x_1), \text{ubc}(x_1), \text{lbc}(x_2), \text{ubc}(x_2)\}
\end{array}
$$

This is easy to see from the definitions of these propagators. If they use $\inf_D x$ then lbc(x) is in the event set, similarly if they use $\sup_D x$ then ubc(x) is in the event set. If they use the entire domain $D(x)$ then dmc(x) is in the event set. □

Indexical propagation solvers [Van Hentenryck et al. 1998; Codognet and Diaz 1996; Carlsson et al. 1997] are based on such reasoning. They define propagators in the form $f(D)(x) = D(x) \cap e(D)$ where e is an indexical expression. The event

set for such propagators is automatically defined by the domain access terms that occur in the expression e.

Example 5.6 *Indexical.* An example of an indexical to propagate $x_1 \geq x_2 + 1$ is

$$
\begin{aligned}
x_1 &\in [\inf(x_2) + 1 ..\ +\infty] \\
x_2 &\in [-\infty ..\ \sup(x_1) - 1]
\end{aligned}
$$

These range expressions for indexicals define two propagators:

$$
\begin{aligned}
f_{\mathrm{i}}(D)(x_1) &= D(x_1) \cap [\inf(x_2) + 1 ..\ +\infty] \\
f_{\mathrm{i}}(D)(x) &= D(x) & x \neq x_1 \\
f_{\mathrm{j}}(D)(x_2) &= D(x_2) \cap [-\infty ..\ \sup(x_1) - 1] \\
f_{\mathrm{j}}(D)(x) &= D(x) & x \neq x_2
\end{aligned}
$$

The event set for the propagator f_{i} from the definition is $\{\mathrm{lbc}(x_2)\}$ while the event set for f_{j} is $\{\mathrm{ubc}(x_1)\}$.   □

Using events we can define a much more accurate version of new that only adds propagators for which one of the events in its event set has occurred.

$$
\mathsf{new}_{\mathrm{events}}(f, F, D, D') = \{f' \in F \mid \mathrm{evset}[f'] \cap \mathrm{events}(D, D') \neq \emptyset\} - \mathrm{fix}(f, D)
$$

This version of new (without dynamic fixpoint reasoning) roughly corresponds with what most constraint propagation systems currently implement.

PROPOSITION 5.7. $\mathsf{new}_{\mathrm{events}}$ *maintains the invariant* $f(D) = D$ *for all* $f \in F - Q$ *at the start of the* **while** *loop.*

PROOF. Consider $f' \in F - Q - \{f\}$ different from the selected propagator f. Then $f'(D) = D$ and if $f'(D') \neq D'$ then $\mathrm{events}(D, D') \cap \mathrm{es}(f') \neq \emptyset$ by case (b) of the definition of $\mathrm{es}(f')$, so $f' \in Q$ at the start of the loop.

Consider the selected propagator f. It is removed from Q, but if $f(D') \neq D'$ then $\mathrm{events}(D, D') \cap \mathrm{es}(f) \neq \emptyset$ by case (a) of the definition of $\mathrm{es}(f)$. Clearly, also $\mathrm{fix}(f, D) \neq \emptyset$. So $f \in Q$ at the start of the loop.   □

5.2   Event Set Experiments

Table V shows runtime and number of propagation steps for different event sets relative to a propagation engine not using events (the engine uses dynamic fixpoint reasoning but no priorities).

General observations. A first, quite surprising, observation is that using no events at all is not so bad. It is the best approach for 10 out of the 25 benchmarks and for `crowded-chess-7` by a considerable margin (between 8.3% and 38.7%). Another general observation is that a reduction in the number of propagation steps does not directly translate into a reduction in runtime. This is due to the fact that all saved propagator executions are cheap: the propagator is already at fixpoint and does not have to perform propagation.

Event set observations. Adding the fix event is particularly beneficial for benchmarks with many disequalities, particularly `queens-400` which only uses disequalities.

Table V. Event set experiments (runtime and propagation steps).

Example	only fix, dmc		with bc		with lbc, ubc	
	time	steps	time	steps	time	steps
all-interval-500	+0.6%	±0.0%	+1.0%	±0.0%	+2.0%	±0.0%
alpha	−2.0%	−4.4%	−7.3%	−19.2%	−6.3%	−19.3%
bibd-7-3-60	−2.6%	−16.6%	−1.4%	−16.6%	+7.4%	−15.8%
cars	+1.1%	−0.2%	+1.7%	−0.2%	+2.6%	−0.2%
crowded-chess-7	+8.3%	±0.0%	+20.2%	−8.4%	+38.7%	−8.4%
donald-b	+0.3%	±0.0%	−2.1%	−9.2%	−1.3%	−9.2%
donald-d	±0.0%	±0.0%	+0.9%	±0.0%	−0.1%	±0.0%
donald-v	+0.6%	−10.5%	−1.4%	−19.7%	+0.1%	−19.7%
golomb-10-b	−0.6%	±0.0%	±0.0%	−3.9%	−0.5%	−3.9%
golomb-10-d	+0.2%	±0.0%	+26.4%	−3.5%	+26.5%	−3.5%
graph-color	−0.7%	−47.7%	+0.1%	−49.5%	+0.6%	−49.5%
grocery	−0.1%	±0.0%	±0.0%	±0.0%	±0.0%	±0.0%
knights-10	−12.7%	−47.5%	−10.8%	−48.6%	−7.7%	−48.6%
minsort-200	+0.4%	±0.0%	+1.1%	±0.0%	+1.5%	±0.0%
o-latin-7-d	−1.2%	±0.0%	+0.2%	−1.7%	+1.5%	−1.7%
partition-32	+0.7%	±0.0%	+15.1%	+3.4%	+18.8%	+17.7%
photo	+1.6%	±0.0%	−3.3%	−27.1%	−2.0%	−26.9%
picture	+2.0%	±0.0%	+3.5%	±0.0%	+11.5%	±0.0%
queens-400	−87.5%	−99.1%	−87.5%	−99.1%	−87.5%	−99.1%
queens-400-a	−10.4%	−38.8%	−9.6%	−38.8%	−8.2%	−38.8%
sequence-500	+2.2%	±0.0%	+3.5%	+36.3%	+8.8%	+36.3%
square-5-d	+0.8%	±0.0%	+2.7%	±0.0%	+2.6%	+0.5%
square-7-b	−0.2%	+0.5%	+0.6%	−8.2%	+0.7%	−7.8%
square-7-v	−1.0%	−8.9%	−1.7%	−16.9%	+1.2%	−16.9%
warehouse	+2.2%	+0.2%	+3.3%	−7.6%	+4.2%	−6.6%
average (all)	−8.5%	−24.3%	−6.7%	−26.9%	−4.6%	−26.4%

Adding the bc event has significant benefit (up to 5%) when there are linear equalities as in `alpha` or bounds($\mathbb{Z}$) consistent `alldifferent` as in `photo`. But for other examples, where one would expect some reduction in runtime, the overhead (to be discussed in more detail below) for maintaining a richer event set exceeds the gains from reducing the number of propagation steps. This is true for examples such as `minsort-200` (minimum propagators), `partition-32` (multiplication), and `golomb-10-b` and `square-7-b` (linear equations and bounds($\mathbb{Z}$) consistent `alldifferent`).

Splitting the bc event into lbc and ubc events exposes the overhead once more. There is almost never an improvement in number of propagations, since only inequalities can actually benefit, and there is substantial overhead.

Influence of propagation order. Similar to using fixpoint reasoning, the use of events also changes the order in which propagators are executed. Table VI reconsiders all examples that could possibly benefit from bc or lbc, ubc events and all examples that show a remarkable increase in number of propagation steps in Table V. The row "average (above)" gives the geometric mean of the relative numbers given in the table whereas "average (all)" shows the relative numbers for all examples.

Table VI.   Event set experiments with priorities (runtime and propagation steps).

Example	only fix, dmc		with bc		with lbc, ubc	
	time	steps	time	steps	time	steps
bibd-7-3-60	+6.1%	−3.3%	+0.6%	−3.3%	+13.1%	−2.5%
crowded-chess-7	+8.0%	±0.0%	+20.4%	−0.7%	+38.0%	−0.7%
donald-b	+0.9%	±0.0%	−1.1%	−9.2%	−0.7%	−9.2%
donald-d	−0.3%	±0.0%	+0.1%	±0.0%	−0.2%	±0.0%
golomb-10-b	+0.1%	±0.0%	−0.9%	−5.5%	−0.9%	−5.5%
golomb-10-d	+0.1%	±0.0%	−0.5%	−7.8%	±0.0%	−7.8%
minsort-200	+0.9%	±0.0%	+1.5%	±0.0%	+1.8%	±0.0%
partition-32	+1.9%	−1.2%	+1.8%	−1.8%	+1.0%	−0.4%
picture	+2.6%	±0.0%	+3.7%	±0.0%	+1.6%	±0.0%
square-5-d	+0.9%	±0.0%	+3.2%	±0.0%	+3.2%	+0.2%
square-7-b	+0.4%	−0.1%	+0.2%	−10.6%	+1.2%	−10.6%
average (above)	+1.9%	−0.4%	+2.5%	−3.6%	+4.8%	−3.4%
average (all)	−7.8%	−24.1%	−7.8%	−27.8%	−6.3%	−27.7%

Table VII.   Event set experiments with priorities (allocated memory).

Example	no events mem (KB)	only fix, dmc mem	with bc mem	with lbc, ubc mem
bibd-7-3-60	6 690.1	+2.9%	+5.7%	+14.4%
crowded-chess-7	198.2	+4.0%	+13.1%	+21.2%
donald-b	7.8	±0.0%	+12.8%	+12.8%
donald-d	3.4	±0.0%	+58.2%	+58.2%
donald-v	5.8	+34.3%	+34.3%	+51.5%
golomb-10-b	39.7	+2.6%	+10.1%	+12.7%
golomb-10-d	37.7	±0.0%	+2.8%	+16.0%
minsort-200	32 419.1	+1.1%	+2.3%	+17.4%
partition-32	160.3	+3.7%	+9.4%	+17.5%
picture	451.6	+17.7%	+17.7%	+30.1%
queens-400	30 286.6	+0.2%	+0.2%	+0.2%
queens-400-a	567.0	+2.8%	+4.2%	+4.2%
average (above)	—	+5.4%	+13.3%	+20.3%
average (all)	—	+3.9%	+9.9%	+15.5%

The numbers confirm that with priorities no considerable increase in runtime can be observed for all but crowded-chess-7, where the increase in runtime here is due to a change in propagation order to the introduction of event sets that does not depend on the relative priorities of the propagators used.

Once priorities are used, lbc and ubc are basically never beneficial.

Memory requirements. Table VII shows the total memory allocated for different event sets relative to a propagation engine not using events (the engine uses dynamic fixpoint reasoning and priorities). It is important to note that the memory figures reflect the amount of memory allocated which is bigger than the amount of memory actually used. In particular, for small examples such as donald-* the increase in allocated memory just reflects the fact that an additional memory block gets allocated.

Using just a fix event increases the required memory by less than 4% in average, while using full event sets increases the required memory by 15.5% in average.

It must be noted that Gecode (the system used) has a particularly efficient implementation of event sets: the implementation uses a single pointer for an entry in an event set; entries for the same variable and the same propagator but with different events still use a single pointer. The memory overhead is due to the fact that per variable and supported event, one single pointer for bookkeeping is needed. Hence, the highest overhead can be expected for examples with many variables but relatively few propagators and small event sets (such as `bibd-7-3-60`, `crowded-chess-7`, and `picture`). The overhead becomes less noticeable for examples with many propagators or large event sets and few variables (such as `queens-400` and `queens-400-a`).

Summary. In summary, while there is a compelling argument for fix events, there is only a weak case for bc being supported, and lbc and ubc should not be used.

5.3 Dynamic Event Sets

Events help to improve the efficiency of a propagation-based solver. Just as we can improve the use of fixpoint reasoning by examining the dynamic case, we can also consider dynamically updating event sets as more information is known about the variables in the propagator.

Monotonic event sets

Definition 5.8 *Monotonic propagator dependence.* A propagator f is *monotonically dependent on* a set of events $\mathrm{es}(f, D)$ *in the context of domain D* iff

(a) for domains $D_0 \sqsubseteq D$: if $f(D_0) \neq f(f(D_0))$ then $\mathrm{events}(D_0, f(D_0)) \cap \mathrm{es}(f, D) \neq \emptyset$,

(b) for domains D_0 and D_1 where $D_0 \sqsubseteq D$, $f(D_0) = D_0$, and $D_1 \sqsubseteq D_0$: if $f(D_1) \neq D_1$ then $\mathrm{events}(D_0, D_1) \cap \mathrm{es}(f, D) \neq \emptyset$.

Clearly given this definition $\mathrm{es}(f, D)$ is monotonically decreasing with D. The simplest kind of event reduction occurs by subsumption.

Definition 5.9 *Subsumption.* A propagator f is *subsumed* for domain D, if for each domain $D' \sqsubseteq D$ we have $f(D') = D'$.

A subsumed propagator makes no future contribution. If f is subsumed by D then $\mathrm{es}(f, D) = \emptyset$ and f is never re-applied. Most current constraint propagation systems take into account propagator subsumption.

Example 5.10 *Subsumption.* Consider the propagator f_a and the domain D with $D(x_1) = [1 .. 3]$ and $D(x_2) = [3 .. 7]$. Then the constraint holds for all $D' \sqsubseteq D$ and $\mathrm{es}(f, D) = \emptyset$. ☐

Changing event sets can occur in cases other than subsumption.

Example 5.11 *Minimum propagator.* A propagator f_k for $x_0 = \min(x_1, x_2)$ can be defined as

$$
\begin{aligned}
f_\mathrm{k}(D)(x_0) &= D(x_0) \cap [\min(\inf_D x_1, \inf_D x_2) .. \min(\sup_D x_1, \sup_D x_2)] \\
f_\mathrm{k}(D)(x_i) &= D(x_i) \cap [\inf_D x_0 .. +\infty] & i \in \{1, 2\} \\
f_\mathrm{k}(D)(x) &= D(x) & x \notin \{x_0, x_1, x_2\}
\end{aligned}
$$

The static event set $es(f_k)$ is $\{lbc(x_0), lbc(x_1), ubc(x_1), lbc(x_2), ubc(x_2)\}$.

But given domain D where $D(x_0) = [1 .. 3]$ and $D(x_2) = [5 .. 7]$ we know that modifying the value of x_2 will never cause propagation. A minimal definition of $es(f_k, D)$ is $\{lbc(x_0), lbc(x_1), ubc(x_1)\}$.　□

Example 5.12 `exactly` *propagator.* The constraint `exactly`$([x_1, \ldots, x_n], m, k)$ holds if exactly m out of the variables $x_1, \ldots, x_n$ are equal to k [Van Hentenryck et al. 1992]. As soon as one of the x_i becomes different from k, all events for x_i can be ignored. Originally the events are $dmc(x_i), 1 \le i \le m$, $lbc(m)$, $ubc(m)$ and $dmc(k)$.

Suppose a domain D where $D(k) = \{1, 3, 8\}$ and $D(x_3) = \{2, 5, 6, 10, 11, 12\}$, then $x_3 \ne k$ and we know its contribution to the `exactly` constraint. We can remove the event $dmc(x_3)$ from the event set safely.　□

Other examples for monotonic event sets are propagators for the `lex` constraint and the generalized `element` constraint. When using a variant of the `lex` propagator proposed in [Carlsson and Beldiceanu 2002], events can be removed as soon as the order among pairs of variables being compared can be decided. For the generalized `element` constraint [Chamard et al. 1995] where the array elements are variables, events for a variable from the array can be safely removed as soon as the variable becomes known to be different from the result of the `element` constraint.

Using monotonic dynamic event sets we can refine our definition of `new` as follows.

$\quad$ $new_{\mathrm{mevents}}(f, F, D, D')$
$\quad\quad$ $F' := \{f' \in F \mid evset[f'] \cap events(D, D')\} - fix(f, D)$
$\quad\quad$ $evset[f] := es(f, D')$
$\quad\quad$ **return** F'

Every time a propagator f is applied its event set is updated to take into account newly available information.

A related idea is the "type reduction" of [Savéant 2000] where propagators are improved as more knowledge on domains (here called types) becomes available. For example, the implementation of $x_0 = x_1 \times x_2$ will be replaced by a more efficient one, when all elements in $D(x_1)$ and $D(x_2)$ are non-negative. Here we concentrate on how the event sets change. The two ideas could be merged as they are complementary.

PROPOSITION 5.13. new_{mevents} *maintains the invariant* $f(D) = D$ *for all* $f \in F - Q$ *at the start of the* ***while*** *loop.*

PROOF. The proof is almost identical to that for Proposition 5.7 since we are working in a context if $evset[f] = es(f, D^*)$ then $D \sqsubseteq D^*$.

For propagators using the monotonic event sets, we have the invariant that $f \in F - Q$ iff $evset[f] = es(f, D^*)$ where D^* is the result of the last time we executed propagator f.

Suppose we have $f' \in F - Q - \{f\}$ where $f'(D') \ne D'$ and $f(D) = D$. Then $events(D^*, D') \cap es(f, D^*) \ne \emptyset$ and since $f' \notin Q$ we have that $events(D^*, D) \cap es(f, D^*) = \emptyset$. Otherwise, we would have placed f in the queue already, hence by the Equation (1) $events(D, D') \cap es(f, D^*) \ne \emptyset$ so $f' \in Q$ at the start of the loop.

Consider the selected propagator f. It is removed from Q but if $f(D') \neq D'$ then clearly $\mathrm{fix}(f, D) = \emptyset$ and using case (a) of Definition 5.8 we have that $\mathrm{events}(D, D') \cap \mathrm{es}(f, D) \neq \emptyset$. Hence $f \in Q$ at the start of the loop. $\square$

Fully dynamic event sets. Note that for many propagators we can be more aggressive in our definition of event sets if we allow the event sets to change in a manner that is not necessarily monotonically decreasing.

Definition 5.14 *General propagator dependence.* A propagator f is *dependent on* a set of events $\mathrm{es}(f, D)$ in the context of domain D if for all domains D_1 where $D_1 \sqsubset D$ and $f(D_1) \neq D_1$ then $\mathrm{events}(D, D_1) \cap \mathrm{es}(f, D) \neq \emptyset$.

Using fully dynamic event sets we can refine our definition of new as follows.

$\mathsf{new}_{\mathrm{devents}}(f, F, D, D')$
$\quad F' := \{f' \in F \mid \mathrm{evset}[f'] \cap \mathrm{events}(D, D')\} - \mathrm{fix}(f, D)$
$\quad \textbf{if } (\mathrm{fix}(f, D) = \emptyset)$
$\quad\quad F' := F' \cup \{f\}$
$\quad \mathrm{evset}[f] := \mathrm{es}(f, D')$
$\quad \textbf{return } F'$

Every time a propagator f is applied its event set is updated to take into account newly available information. The only difficult case is that if the definition of dynamic dependency does not capture the events that occur when moving from D to $D' = f(D)$. If fixpoint reasoning cannot guarantee a fixpoint we need to add f to the queue.

Fully dynamic event sets are more powerful than monotonically decreasing event sets, but in general they require reasoning about the new event sets each time the propagator is run.

Example 5.15 *Fully dynamic events for minimum.* Given the domain D where $D(x_0) = [0 .. 10]$, $D(x_1) = [0 .. 15]$, and $D(x_2) = [5 .. 10]$ and the propagator f_{k} from Example 5.11. D is a fixpoint of f_{k} and a minimal set $\mathrm{es}(f_{\mathrm{k}}, D)$ is

$$\{\mathrm{lbc}(x_0), \mathrm{lbc}(x_1), \mathrm{ubc}(x_1), \mathrm{ubc}(x_2)\}$$

While at $D' \sqsubseteq D$ where $D'(x_0) = [5 .. 9]$, $D'(x_1) = [6 .. 9]$, and $D'(x_2) = [5 .. 10]$, which is also a fixpoint, the minimal set $\mathrm{es}(f_{\mathrm{k}}, D')$ is

$$\{\mathrm{lbc}(x_0), \mathrm{ubc}(x_1), \mathrm{lbc}(x_2), \mathrm{ubc}(x_2)\}$$

For the constraint c_{k} we simply need to maintain a lbc event for some variable x_i in the right hand side with the minimal lbc value. $\square$

Example 5.16 *Fully dynamic events for Boolean inequality constraints.* Assume a propagator f_{l} for the constraint $\sum_{i=1}^{n} x_i \geq k$ over Boolean variables x_i, $1 \leq i \leq n$, and the domain D where $D(x_i) = \{0, 1\}$, $1 \leq i \leq n$. This propagator only propagates when $n - k$ variables are set to 0 in which case the remainder are set to 1. A minimal dynamic event set $\mathrm{es}(f_{\mathrm{l}}, D)$ is

$$\{\mathrm{fix}(x_j) \mid j \in S\}$$

for any $S \subseteq [1 .. n]$ where $|S| > k$. That is, only $k+1$ variables must be watched: if none of the $k + 1$ variables is fixed then it cannot be that $n - k$ variables are fixed to 0.

Suppose a variable x_j with $j \in S$ is fixed to 1. Then the propagator still cannot fire even if all the variables $x_{j'}$ with $j \notin S$ are fixed to 0, so the dynamic event set need not change.

Suppose a variable x_j with $j \in S$ is fixed to 0. Then the propagator needs to find another variable $x_{j'}$ with $j' \notin S$ where $D(x_{j'}) = [0 .. 1]$ and replaces $\text{fix}(x_j)$ in the dynamic event set by $\text{fix}(x_{j'})$. $\qquad\square$

PROPOSITION 5.17. $\text{new}_{devents}$ *maintains the invariant* $f(D) = D$ *for all* $f \in F - Q$ *at the start of the* **while** *loop.*

PROOF. We have the invariant that $f \in F - Q$ iff $\text{evset}[f] = \text{es}(f, D^*)$ where D^* is the result of the last time we executed propagator f.

Suppose we have $f' \in F - Q - \{f\}$ where $f'(D') \neq D'$ and $f(D) = D$ then $\text{events}(D^*, D') \cap \text{es}(f, D^*) \neq \emptyset$ and since $f' \notin Q$ we have that $\text{events}(D^*, D) \cap \text{es}(f, D^*) = \emptyset$ otherwise we would have placed f' in the queue already, hence by the equation (1) $\text{events}(D, D') \cap \text{es}(f, D^*) \neq \emptyset$ so $f' \in Q$ at the start of the loop.

Consider the selected propagator f. The same reasoning cannot apply since even though we know that $\text{events}(D^*, D') \cap \text{es}(f, D^*) \neq \emptyset$, we have no guarantee that $\text{events}(D, D') \cap \text{es}(f, D^*)$ is not empty. Now if $\text{fix}(f, D) = \emptyset$ then possibly $f(D') \neq D'$, but this will force $f \in Q$ by the start of the loop. $\quad\square$

Effectively, the fully dynamic event sets approach relies only on the dynamic fixpoint reasoning of the propagator f to handle what happens when moving from D to D'.

Fully dynamic event sets are closely related to the watched literals approach to improving unit propagation in SAT solving [Moskewicz et al. 2001]. A clause in CNF corresponds to the constraint $\sum_{i=1}^{n} x_i \geq 1$ explored in Example 5.16. Using watched literals, unit propagation only considers a clause for propagation if one of two *watched* literals in the clause becomes false. Recently, the idea of watched literals has been used in constraint programming for the Minion solver [Gent et al. 2006]. Watched literals differ from the events we concentrate on here since they take into account values (similar to the $\text{neq}(x, a)$ event) and require the knowledge of which variable is changed by propagation. Note that dynamic event sets do not usually have the property of watched literals, in that they do not need to be updated on backtracking. The setup chosen for propagation in this paper is propagator-centered: the incremental propagation solver in Figure 2 organizes propagation by maintaining the set of propagators Q. Information about which variables change and how their domain changes during propagation as needed for watched literals requires additional effort, see [Lagerkvist and Schulte 2007].

5.4 Dynamic Event Sets Experiments

Table VIII shows the comparison of monotonic and fully dynamic event sets to a propagation solver using static event sets with $\{\text{fix}, \text{bc}, \text{dmc}\}$ events and priorities to avoid priority inversion as discussed before. The table lists only examples where dynamic event sets are used.

Table VIII. Dynamic event sets experiments (with priorities).

Example	monotonic			fully dynamic		
	time	steps	mem	time	steps	mem
`bibd-7-3-60`	$+0.4\%$	$\pm0.0\%$	$\pm0.0\%$	-5.0%	-5.2%	$\pm0.0\%$
`crowded-chess-7`	-24.0%	$\pm0.0\%$	-15.2%	-28.2%	-34.2%	-17.8%
`sequence-500`	-82.3%	-78.3%	-49.5%	-82.1%	-78.3%	-49.5%
`o-latin-7-d`	-0.7%	$\pm0.0\%$	$\pm0.0\%$	-1.7%	$\pm0.0\%$	$\pm0.0\%$
average (above)	-39.5%	-31.8%	-19.1%	-41.1%	-39.3%	-19.7%
average (all)	-10.8%	-8.2%	-4.1%	-11.2%	-9.9%	-4.2%

The propagators using monotonic event sets are as follows: for `bibd-7-3-60`:
`lex`; for `crowded-chess-7`: `exactly` and `element`; for `sequence-500`: `exactly`;
for `o-latin-7-d`: `lex`. Clearly, monotonic event sets lead to a drastic reduction in
both runtime and memory usage, where it is worth noting that the reduction in time
is even more marked than the reduction in propagation steps (each propagation step
becomes cheaper as smaller event sets must be maintained).

Fully dynamic event sets are considered in Boolean-sum propagators used in
`bibd-7-3-60` and `crowded-chess-7`. The difference in improvement between the
two examples can be explained by the fact that `crowded-chess-7` uses Boolean-
sums as inequalities while `bibd-7-3-60` uses Boolean-sums as equalities where in-
equalities offer the potential for considerably smaller event sets [Gent et al. 2006].

Example `bibd-7-3-60` provides another insight: the drastic reduction in runtime
observed in [Gent et al. 2006] by using watched literals for Boolean-sum propagators
in the Minion solver is most likely not due to small event sets but to other aspects.
A possible aspect is the knowledge about which variables have been modified when
a propagator is executed.

6. WHICH PROPAGATOR TO EXECUTE NEXT

We now address how to define which propagator f in the queue Q should execute
first, that is, how to define the choose function.

The simplest policy to implement is a FIFO (First In First Out) queue of prop-
agators. Propagators are added to the queue, if they are not already present, and
choose selects the oldest propagator in the queue. The FIFO policy ensures fair-
ness so that computation is not dominated by a single group of propagators, while
possibly not discovering failure (a false domain) from other propagators quickly.

The equally simple LIFO (Last In First Out) policy is a stack where propagators
not already in the stack are pushed, and choose selects the top of the stack.

6.1 Basic Queuing Strategy Experiments

Table IX compares using a FIFO queue versus a LIFO stack. The result, according
with folklore knowledge, clearly illustrates that a queue must be used. The few
cases where a stack is better are outweighed by the worst cases for a stack.

Later experiments in Section 6.3 using priorities with combinations of FIFO
queues and LIFO stacks reveal that the pathological behavior of `all-interval-500`
is due to priority inversion. However, the pathological behavior of `minsort-200` is
due to the use of a LIFO stack.

Table IX.   Queue versus stack experiments.

Example	queue		stack	
	time (ms)	steps	time	steps
`all-interval-500`	73.00	377 777	+13626.0%	+32.2%
`alpha`	97.31	207 470	+47.5%	+99.0%
`bibd-7-3-60`	2 020.30	1 020 162	+7.7%	+102.7%
`cars`	4.65	14 860	+1.3%	+2.6%
`crowded-chess-7`	553.25	660 525	+4.7%	+68.1%
`donald-b`	0.62	414	+36.3%	+31.9%
`donald-d`	28.53	40	−5.4%	+7.5%
`donald-v`	0.36	366	+7.6%	+23.2%
`golomb-10-b`	1 342.48	2 095 161	+870.6%	+171.5%
`golomb-10-d`	3 201.84	2 091 691	−11.9%	+98.9%
`graph-color`	33.54	4 422	+299.7%	+118.4%
`grocery`	56.87	2 211	+44.2%	−6.3%
`knights-10`	6.66	25 225	+7.7%	+53.3%
`minsort-200`	141.84	113 437	+1580.1%	+1123.3%
`o-latin-7-d`	532.48	311 135	+4.9%	+24.9%
`partition-32`	9 001.24	13 939 101	−18.5%	+34.8%
`photo`	90.37	296 661	+11.1%	+8.7%
`picture`	1 583.12	119 406	+27.1%	+79.9%
`queens-400`	549.36	268 771	±0.0%	−0.1%
`queens-400-a`	14.40	1 265	−0.9%	−0.3%
`sequence-500`	98.12	56 048	+53.8%	+240.9%
`square-5-d`	32 263.12	1 478 403	+71.5%	+57.7%
`square-7-b`	9 491.84	6 054 656	+75.3%	+44.9%
`square-7-v`	5 345.00	9 636 675	+8.3%	+35.2%
`warehouse`	0.70	2 075	+11.7%	+29.3%
average (all)	—	—	+78.4%	+61.0%

6.2   Static Priorities

A statically prioritized queue associates with each propagator a fixed priority, we will assume an integer in the range $[0 \,..\, k-1]$. In effect, the queue Q is split into k queues, $Q[0], \ldots Q[k-1]$ where each $Q[i]$ is a FIFO queue for the propagators with priority i. Selection always chooses the oldest propagator in the lowest numbered queue $Q[i]$ that is non-empty. Static prioritization allows one to ensure that quick propagators are executed before slow propagators.

We give an example of seven static priorities, with names of the integer priorities as follows: `unary`=0, `binary`=1, `ternary`=2, `linear`=3, `quadratic`=4, `cubic`=5, and `veryslow`=6. The names are meant to represent the arity of the constraint (that is, the number of its variables), and then the asymptotic runtime of the propagator, once the constraint can handle n variables. So `binary` is for binary constraints, `quadratic` is for constraints that are approximately $O(n^2)$ for instances with n variables.

Example 6.1 *Propagator priorities.* The propagator f_{m} for $\mathrm{even}(x_1)$ defined by

$$f_{\mathrm{m}}(D)(x_1) \;=\; D(x_1) \cap \left[2\lceil \tfrac{1}{2}\inf_D x_1 \rceil \,..\, 2\lfloor \tfrac{1}{2}\sup_D(x_1) \rfloor \right]$$
$$f_{\mathrm{m}}(D)(x) \;=\; D(x) \qquad\qquad\qquad\qquad\qquad\qquad x \neq x_1$$

might be given priority `unary`, while f_e and f_f might be given priority `binary`.

The domain propagator defined in [Régin 1994] for the `alldifferent` constraint $\wedge_{i=1}^n \wedge_{j=i+1}^n x_i \neq x_j$ (with complexity $O(n^{2.5})$) might be given priority `quadratic`.

The `alldifferent` bounds($\mathbb{Z}$) propagator defined in [Puget 1998] (with complexity $O(n \log n)$) might be given priority `linear`. □

Priorities in effect force many more fixpoints to be calculated. A fixpoint of all propagators at priority level i and lower must be reached before a propagator at priority level $i + 1$ is run. With other words, whenever an expensive propagator at priority level $i + 1$ is executed and changes the domain, cheaper propagators at priority levels i and lower must be re-executed to compute a fixpoint. This means we will often cause more propagators to run when using priorities, but more cheap propagators!

Example 6.2 *Repeated fixpoints.* Consider the execution of a system of idempotent propagators for constraints $c_c \equiv x_1 = 2x_2$, $c_d \equiv x_1 = 3x_2$, $c_n \equiv x_2 \leq 6 \rightarrow x_1 \leq x_3 + 7$ and $c_o \equiv \mathtt{alldifferent}([x_1, x_2, x_3, x_4, x_5])$. We will use the idempotent bounds($\mathbb{R}$) propagators f_c^*, f_d^*, and f_n^* for the first three constraints, and the domain propagator, f_o, for `alldifferent` from [Régin 1994], for the last constraint. Let the initial domain be $D(x_1) = [0 .. 18]$, $D(x_2) = [0 .. 9]$, $D(x_3) = [0 .. 6]$, and $D(x_4) = D(x_5) = [0 .. 3]$. The priorities are `binary`, `binary`, `ternary`, and `quadratic` respectively. All propagators are at fixpoint.

Suppose the domain of x_1 changes to $[0 .. 17]$. All propagators are enqued. The priority level `binary` propagators are run to fixpoint (as in Example 4.4) giving $D(x_1) = [0 .. 12]$, $D(x_2) = [0 .. 6]$, $D(x_3) = [0 .. 4]$ Then f_n^* is scheduled and causes $D(x_1) = [0 .. 11]$. Both f_c^* and f_d^* are enqued, and after executing f_c^*, f_d^*, f_c^* and f_d^* the next fixpoint of the `binary` priority propagators is reached: $D(x_1) = [0 .. 6]$, $D(x_2) = [0 .. 3]$, $D(x_3) = [0 .. 2]$. Then f_n^* is scheduled again and causes no change. After that, f_o is executed, it reduces the domains of $D(x_1) = [4 .. 6]$ since all the values $[0 .. 3]$ are required for the variables x_2, x_3, x_4, x_5. Both f_c^* and f_d^* are enqued, and after executing we reach their fixpoint $D(x_1) = \{6\}$, $D(x_2) = \{3\}$, $D(x_3) = \{2\}$. Once again f_n^* is executed for no change. Then f_o is executed once more obtaining $D(x_4) = D(x_5) = \{0, 1\}$. This is the overall fixpoint. □

We can adjust the granularity of the priorities: we can have a finer version of the above priorities with 14 priorities, each priority above with a `low` and `high` version. This allows us to separate, for example, a domain consistent propagator for the binary absolute value constraint $\mathrm{abs}(x) = y$ (`binary-low`) from a bounds consistent propagator for the same constraint (`binary-high`). This increased granularity will be important in Section 7.

Conversely, we may collapse the priorities into fewer levels, for example into three levels `unary`↔`ternary`, `linear`↔`quadratic`, `cubic`↔`veryslow`.

Another model for priorities in constraint propagation based on composition operators is [Granvilliers and Monfroy 2003]. The model, however, runs all propagators of lower priority before switching propagation back to propagators of higher priority. This model does not preempt computing a fixpoint for a low priority. The model always completes a fixpoint for a given priority level and only then possibly continues at a higher priority level.

Most systems have some form of static priorities, typically using two priority levels (for example, SICStus [Intelligent Systems Laboratory 2004], Mozart [Mozart Consortium 1999]). The two levels are often not entirely based on cost: in SICStus all indexicals have high priority and all other lower priority. While ECL^{i}PS^e [Wallace et al. 1997; Harvey 2004] supports twelve priority levels, its finite domain solver also uses only two priority levels where another level is used to support constraint debugging.

A similar, but more powerful approach is used by Choco [Laburthe 2000] using seven priority levels allowing both LIFO and FIFO traversal.

Prioritizing particular operations during constraint propagation is important in general. For interval narrowing, prioritizing constraints can avoid slow convergence, see for example [Lhomme et al. 1998].

The prioritizing of propagators by *cost* is important, inverting the priorities can lead to significant disadvantages.

Example 6.3 *Inverted priorities.* Consider executing Example 6.2 with inverted priorities. We first execute f_o then f_n^* for no effect. Then executing f_c^* modifies $D(x_1)$, so each of f_o and f_n^* are enqued and re-executed for no effect. Then executing f_d^* has the same behavior. Overall we execute the propagators f_n^* and f_o each at least 10 times, as opposed to 3 and 2 times respectively in Example 6.2. Since they are the most expensive to execute we would expect this to be slower (this is confirmed immediately below).   □

6.3   Static Priority Experiments

Priority granularity. Table X gives runtime and propagation steps of various priority granularities compared to a propagation engine using a FIFO queue (and using dynamic fixpoint reasoning, events of types {fix, bc, dmc}, and fully dynamic event sets). The different experiments capture different priority granularities: "small" uses three priorities (unary↔ternary, linear↔quadratic, cubic↔veryslow); "medium" uses seven priority levels (from unary to veryslow); "full" uses 14 priority levels (from unary-high to veryslow-low).

The results illustrate that even when there are substantially more propagations (for example, golomb-10-{b,d}) there can be significant savings. Priorities can have a very substantial saving (almost 50% for golomb-10-d) and the worst case cost on the benchmarks is only 10.3%. Overall while the "medium" range of priorities is preferable on many benchmarks, the "full" range of priorities gives real speedups on two more benchmarks (photo and square-7-b) and tends to reduce the worst case behavior of "medium".

Examples such as queens-400, queens-400-a, and sequence-500 for "small" only feature propagators with the same priority (the number of propagation steps remains the same). Hence the increase in runtime by less than 1% describes the overhead of using priorities at all.

Depending on the underlying system, increasing the granularity also requires more memory. Gecode, as a system based on recomputation and copying, constitutes the worst case in that each copied node in the search tree maintains queues for all priority levels. However, the average increase in used (not allocated as before) memory is +0.2% for "small", +0.7% for "medium", and +1.5% for "full"

Table X.   Priority experiments with varying granularities.

Example	small		medium		full	
	time	steps	time	steps	time	steps
`all-interval-500`	+1.8%	+0.3%	+2.3%	+0.3%	+2.3%	+0.3%
`alpha`	+1.2%	±0.0%	+1.2%	±0.0%	+1.2%	±0.0%
`bibd-7-3-60`	+0.1%	+16.9%	+0.1%	+16.9%	+0.2%	+16.9%
`cars`	−1.4%	−10.4%	−1.5%	−10.5%	−1.8%	−10.5%
`crowded-chess-7`	+8.1%	+91.2%	+9.0%	+91.2%	+8.8%	+91.1%
`donald-b`	+1.1%	±0.0%	+0.4%	±0.0%	+0.9%	±0.0%
`donald-d`	−0.1%	±0.0%	−5.6%	+5.0%	−5.5%	+5.0%
`donald-v`	+1.6%	±0.0%	+2.6%	±0.0%	+2.2%	±0.0%
`golomb-10-b`	−33.1%	+33.5%	−31.3%	+51.0%	−31.6%	+51.0%
`golomb-10-d`	−49.5%	+29.2%	−49.0%	+46.7%	−48.9%	+46.7%
`graph-color`	−1.3%	±0.0%	−1.3%	−0.1%	−0.9%	−0.1%
`grocery`	+10.3%	−9.2%	−5.6%	+1.9%	−5.6%	+1.9%
`knights-10`	−3.4%	−19.5%	−4.7%	−19.9%	−3.7%	−19.9%
`minsort-200`	+0.5%	±0.0%	+0.5%	±0.0%	−0.2%	±0.0%
`o-latin-7-d`	−10.6%	+4.6%	−11.0%	+4.6%	−10.6%	+4.6%
`partition-32`	−38.6%	−18.2%	−38.7%	−17.8%	−38.4%	−17.8%
`photo`	+0.1%	+3.4%	+0.2%	+3.4%	−7.3%	−1.8%
`picture`	−1.0%	±0.0%	−1.5%	±0.0%	+0.2%	±0.0%
`queens-400`	+0.7%	±0.0%	+0.5%	±0.0%	+0.5%	±0.0%
`queens-400-a`	+0.2%	±0.0%	+0.2%	±0.0%	+0.3%	±0.0%
`sequence-500`	+0.5%	±0.0%	+0.3%	±0.0%	+0.6%	±0.0%
`square-5-d`	−0.5%	±0.0%	+0.9%	+23.8%	+1.1%	+23.8%
`square-7-b`	+1.3%	±0.0%	+0.4%	±0.0%	−23.4%	+22.0%
`square-7-v`	+0.9%	±0.0%	+0.7%	±0.0%	+0.9%	±0.0%
`warehouse`	+5.3%	+13.0%	+5.7%	+12.0%	+5.3%	+12.0%
average (all)	−5.6%	+3.8%	−6.3%	+6.4%	−7.4%	+7.0%

and hence can be neglected.

A broad spectrum of priorities will be useful for the optimizations presented in Section 7. Therefore it is important that while "full" does not offer huge advantages over "medium", it neither degrades overall performance nor requires much memory.

Priorities and stacks. Table XI gives the runtime and propagation steps of using priorities together with stacks or combinations of stacks and queues for the different priority levels. All numbers are given relative to a propagation engine using the "full" priority spectrum with only queues for each priority level. All propagation engines considered also use the full priority spectrum. The propagation engine for "for all" uses only stacks for all priority levels, whereas "for $1, 2, 3$-ary" ("for $1, 2$-ary") uses stacks for priority levels `unary-high` to `ternary-low` (`unary-high` to `binary-low`) and queues for the other levels.

The numbers for "for all" clarify that the misbehavior of LIFO stacks is not due to priority inversion. The folklore belief that LIFO stacks are good for small propagators is refuted by the numbers for "for $1, 2, 3$-ary" and "for $1, 2$-ary". Only three examples show improvement in both cases while `grocery` "for $1, 2, 3$-ary" already exhibits pathological behavior. The measurements show that queue versus stack does not matter for unary or binary constraints whereas stacks are wrong for

Table XI.   Priority experiments with stacks and priorities.

Example	for all		for 1, 2, 3-ary		for 1, 2-ary	
	time	steps	time	steps	time	steps
all-interval-500	−0.6%	±0.0%	+1.3%	±0.0%	+0.2%	±0.0%
alpha	+47.2%	+99.0%	+2.3%	±0.0%	+1.0%	±0.0%
bibd-7-3-60	+5.5%	+49.7%	−4.0%	+1.4%	−5.9%	+1.4%
cars	−4.2%	−7.3%	+1.2%	−2.0%	+1.8%	−2.0%
crowded-chess-7	+0.6%	+10.4%	+2.0%	±0.0%	+1.7%	±0.0%
donald-b	+0.3%	±0.0%	+1.6%	±0.0%	+0.9%	±0.0%
donald-d	−0.6%	+2.4%	+1.0%	±0.0%	+0.3%	±0.0%
donald-v	+7.7%	+23.2%	+2.5%	±0.0%	+1.7%	±0.0%
golomb-10-b	+7.1%	+27.0%	+9.7%	+27.0%	+1.2%	±0.0%
golomb-10-d	+5.7%	+39.2%	+7.7%	+39.2%	+1.0%	±0.0%
graph-color	+62.7%	+77.1%	+2.3%	±0.0%	+1.5%	±0.0%
grocery	+42.8%	+7.1%	+45.8%	+7.1%	+1.6%	−0.3%
knights-10	+0.1%	+5.0%	+2.1%	+5.0%	+1.4%	+5.0%
minsort-200	+1545.4%	+1106.4%	+2.2%	±0.0%	+2.4%	±0.0%
o-latin-7-d	−3.3%	+5.2%	+1.0%	+0.3%	−0.4%	+0.3%
partition-32	−2.2%	−2.9%	+2.2%	+2.3%	+2.2%	+0.4%
photo	−1.3%	−0.8%	−1.0%	−0.9%	+0.1%	−0.9%
picture	+21.1%	+79.9%	−1.3%	±0.0%	−1.2%	±0.0%
queens-400	−0.1%	−0.1%	+1.0%	−0.1%	+0.8%	−0.1%
queens-400-a	−1.0%	−0.3%	+1.2%	±0.0%	+1.2%	±0.0%
sequence-500	+54.1%	+240.9%	+1.4%	±0.0%	+0.8%	±0.0%
square-5-d	+65.2%	+31.6%	+1.2%	±0.0%	+0.8%	±0.0%
square-7-b	+1.7%	+18.6%	+1.4%	±0.0%	+0.8%	±0.0%
square-7-v	+8.9%	+35.2%	+2.9%	±0.0%	+2.6%	±0.0%
warehouse	−2.5%	+1.7%	+3.0%	+0.2%	+3.0%	+0.2%
average (all)	+23.9%	+36.0%	+3.3%	+2.8%	+0.8%	+0.2%

anything else.

Issues to avoid. Table XII shows runtime and propagation steps of using priorities in flawed ways. The experiment "complete fixpoints" refers to the model proposed in [Granvilliers and Monfroy 2003] where fixpoints are always completed before possibly switching to a higher priority level. As to be expected, the number of propagation steps is reduced, however at the expense of increased runtime. More importantly, two examples exhibiting substantial slowdown (golomb-10-d and square-5-d) are particularly relevant as they feature propagators of vastly different priority levels.

The surprising behavior of bibd-7-3-60 appears to be due to a problem with how the priorities for the two different kinds of propagators used in this example (lex and Boolean-sum) are classified. This clarifies that even with a rich spectrum of priority levels at disposal it remains difficult to assign priority levels to propagators.

The experiment "inverse priorities" shows numbers for a propagation engine where a propagator with lowest priority is executed first. The experiment shows that there is a point to the priority levels, high priority = fast propagator. While the number of propagations is often reduced the approach is rarely better than no priorities and sometimes catastrophically worse.

Table XII.   Priority experiments: issues to avoid.

Example	complete fixpoints		inverse priorities	
	time	steps	time	steps
`all-interval-500`	−0.8%	±0.0%	+13296.5%	+32.1%
`alpha`	+0.5%	±0.0%	+0.3%	±0.0%
`bibd-7-3-60`	−11.7%	−45.8%	−7.7%	−42.8%
`cars`	−1.2%	−0.9%	+14.4%	+40.1%
`crowded-chess-7`	−4.2%	−41.1%	−7.4%	−52.0%
`donald-b`	+0.9%	±0.0%	+36.8%	+31.9%
`donald-d`	+0.9%	±0.0%	+6.2%	−7.1%
`donald-v`	+1.6%	±0.0%	+0.2%	±0.0%
`golomb-10-b`	−2.5%	−14.5%	+892.0%	−5.7%
`golomb-10-d`	+14.6%	−14.2%	+945.4%	−2.2%
`graph-color`	+1.9%	+0.1%	+336.8%	+51.1%
`grocery`	+29.3%	−23.4%	+35.3%	+9.1%
`knights-10`	−0.9%	±0.0%	+8.7%	+43.5%
`minsort-200`	+5.5%	+8.6%	+2732.2%	+2246.7%
`o-latin-7-d`	+7.3%	−3.8%	+105.1%	+18.7%
`partition-32`	+0.4%	−1.2%	+108.3%	+66.8%
`photo`	+1.2%	±0.0%	+18.6%	−1.9%
`picture`	−3.6%	±0.0%	−3.7%	±0.0%
`queens-400`	−0.7%	±0.0%	+0.2%	±0.0%
`queens-400-a`	+0.3%	±0.0%	−0.3%	±0.0%
`sequence-500`	+0.2%	±0.0%	−0.2%	±0.0%
`square-5-d`	+32.5%	+7.9%	+50.1%	+7.5%
`square-7-b`	+0.5%	±0.0%	+177.4%	−7.6%
`square-7-v`	+2.6%	±0.0%	±0.0%	±0.0%
`warehouse`	−3.7%	−9.0%	+14.7%	+11.3%
average (all)	+2.5%	−6.6%	+107.5%	+18.4%

The experiment "inverse priorities" clarifies a very important aspect of priorities: they not only serve as a means to improve performance, they also serve as a *safeguard* against pathological propagation order.

6.4   Dynamic Priorities

As evaluation proceeds, variables become fixed and propagators can be replaced by more specialized versions. If a propagator is replaced by a more specialized version, also its priority should change.

Example 6.4 *Updating a propagator.* Consider the propagator f_p for updating x_1 in the constraint $x_1 = x_2 + x_3$ defined by

$$f_\mathrm{p}(D)(x_1) = D(x_1) \cap [\inf_D(x_2) + \inf_D(x_3) .. \sup_D(x_2) + \sup_D(x_3)]$$
$$f_\mathrm{p}(D)(x) = D(x) \qquad\qquad x \neq x_1$$

might have initial priority `ternary`. When the variable x_2 becomes fixed to d_2 say, then the implementation for x_1 can change to

$$f_\mathrm{p}(D)(x_1) = D(x_1) \cap [d_2 + \inf_D(x_3) .. d_2 + \sup_D(x_3)]$$

and the priority can change to `binary`.   □

Table XIII.   Dynamic priority experiments.

Example	dynamic	
	time	steps
`alpha`	-25.5%	-41.7%
`cars`	-5.9%	-13.4%
`crowded-chess-7`	-5.0%	-26.5%
`o-latin-7-d`	-7.3%	-7.4%
`picture`	-1.9%	$\pm 0.0\%$
`sequence-500`	$+6.0\%$	$+34.8\%$
average (above)	-7.1%	-12.1%
average (all)	-1.7%	-3.0%

Changing priorities is also relevant when a propagator with $n > 3$ variables with priority `linear` (or worse) reduces to a binary or ternary propagator.

6.5   Dynamic Priority Experiments

Table XIII shows runtime and propagation steps for an engine using dynamic priorities compared to an engine using all optimizations introduced earlier and the full priority spectrum. Dynamically changing the priority of propagators as they become smaller due to fixed variables can lead to significant improvements. In effect, constraints that become smaller (and thus run at higher priority) are run first causing the still large constraints to be run less often.

This is in particular true for `alpha` with initially only propagators for linear equalities with priority `linear`. When fixing variables during search many of these propagators are then run at priority levels `binary` and `ternary`. It is worth noting that using dynamic priorities can disturb the FIFO queue behavior: for `sequence-500` it appears to be more important to run all `exactly` propagators at the same priority level. Running some of the `exactly` propagators at priorities `binary` and `ternary` is not beneficial and disturbs the queue behavior.

Dynamic priorities incur the overhead to compute the priority based on the number of not yet fixed variables. However, the overhead is still small enough to make dynamic priorities worthwhile overall.

7.   COMBINING PROPAGATION

There are many ways to define a correct propagator f for a single constraint c: the art of building propagators is to find good tradeoffs in terms of speed of execution versus strength of propagation. Typically a single constraint may have a number of different propagator implementations: the cheapest simple propagator, a more complex bounds propagator, and a more complex domain propagator, for example.

Example 7.1 `alldifferent` *propagators*. Consider the propagator $f_q(D)$ for the `alldifferent` constraint.

$$E := \emptyset$$
for $i \in [1 .. n]$
    if $(\exists d.D(x_i) = \{d\})$
        if $(d \in E)$ **return** $D_\perp$ **else** $E := E \cup \{d\}$

> **for** $i \in [1 .. n]$
>   **if** $(|D(x_i)| > 1)$ $D(x_i) := D(x_i) - E$
> **return** D

The propagator does a linear number of set operations in each invocation and is checking. It can be made idempotent by testing that no variable becomes fixed.

Another propagator for the same constraint is the domain propagator f_o introduced in [Régin 1994] with complexity $O(n^{2.5})$. □

Given two propagators, say f_1 and f_2 in prop(c), where f_1 is stronger than f_2 ($f_1(D) \sqsubseteq f_2(D)$ for all domains D), we could choose to implement c by just f_1 or just f_2 trading off pruning versus execution time. Without priorities there is no point in using both propagators, since f_1 will always be run and always compute stronger domains than f_2.

We could possibly merge the implementation of the propagators to create a new propagator $f_{12}(D) = f_1(f_2(D))$. By running the cheaper propagator immediately first we hope that we can (a) quickly determine failure in some cases and (b) simplify the domains before applying the more complicated propagator f_1. While this immediate combination of two propagators in essence is simply building a new propagator, once we have priorities in our propagation engine we can use two or more propagators for the same constraint in different ways.

7.1 Multiple Propagators

Once we have priorities it makes sense to use multiple propagators to implement the same constraint. We can run the weaker (and presumably faster) propagator f_2 with a higher priority than f_1. This makes information available earlier to other propagators. When the stronger propagator f_1 is eventually run, it is able to take advantage from propagation provided by other cheaper propagators.

Note that this is essentially different from having a single propagator f_{12} that always first runs the algorithm of f_2 and then the algorithm of f_1.

Example 7.2 *Multiple* `alldifferent`. Consider the two propagators f_q and f_o defined in Example 7.1 above. We can use both propagators: f_q with priority `linear`, and f_o with priority `quadratic`. This means that we will not invoke f_o until we have reached a fixpoint of f_q and all `linear` and higher priority propagators.

Consider the additional propagator f_e for the constraint $3x_1 = 2x_2$, which has priority `binary`. Consider the domain D where $D(x_1) = \{4, 6\}$, $D(x_2) = \{6, 9\}$, $D(x_3) = \{6, 7\}$ and $D(x_4) = \cdots = D(x_n) = [1 .. n]$, which is a fixpoint for f_e, f_q and f_o. Now assume the domain of x_3 is reduced to 6. Propagator f_q is placed in queue `linear` and f_o is placed in queue `quadratic`. Applying f_q removes 6 from the domain of all the domains of $x_1, x_2, x_4, \ldots, x_n$, and this causes f_e to be placed in queue `binary`. This is the next propagator considered and it causes failure. Propagator f_o is never executed.

If we just use f_o then we need to invoke the more expensive f_o to obtain the same domain changes as f_q, and then fail. □

Multiple propagators for the same constraint have been explored previously in the context of interval narrowing. Monfroy [1990] describes the use of several propagators (narrowing operators) with different propagation strength. In this approach,

after a fixpoint has been computed with cheap propagators, only more expensive propagators are considered (resembling the model based on composition operators discussed in Section 6.2).

7.2 Staged Propagators

Once we are willing to use multiple propagators for a single constraint it becomes worth considering how to more efficiently manage them. Instead of using two (or more) distinct propagators we can combine the several propagators into a single propagator with more effective behavior.

We assume that a propagator has an internal state variable, called its *stage*. When it is invoked, the stage determines what form of propagation applies.

Example 7.3 *Staged* `alldifferent`. Consider the `alldifferent` constraint with implementations f_q and f_o discussed in Example 7.1. We combine them into a staged propagator as follows:

— On a fix(x) event, the propagator is moved to stage A, and placed in the queue with priority `linear`.

— On a dmc(x) event, unless the propagator is in stage A already, the propagator is put in stage B, and placed in the queue with priority `quadratic`.

— Execution in stage A uses f_q, the propagator is put in stage B, and placed in the queue with priority `quadratic`, unless it is subsumed.

— Execution in stage B uses f_o, afterwards the propagator is removed from all queues (stage NONE).

The behavior of the staged propagator is identical to the multiple propagators for the sample execution of Example 7.1. In addition to the obvious advantage of having a single staged propagator, another advantage comes from avoiding the execution of f_o when the constraint is subsumed.   □

In addition to giving other propagators with higher priority the opportunity to run before the expensive part of a staged propagator, the first stage of a propagator can already determine that the next stage does not need to be run. This is illustrated by the following example.

Example 7.4 *Staged linear equations*. Consider the unit coefficient linear equation $\Sigma_{i=1}^{n} a_i x_i = d$ constraint where $|a_i| = 1, 1 \leq i \leq n$. We have two implementations, f_r, which implements bounds($\mathbb{R}$) consistency (considering real solutions, with linear complexity) for the constraint, and f_s, which implements domain consistency (with exponential complexity).

We combine them into a staged propagator as follows:

— On a bc(x) (or depending on the event types available: lbc(x) or ubc(x)) event, the propagator is moved to stage A, and placed in the queue with priority `linear`.

— On a dmc(x) event, unless the propagator is in stage A already, the propagator is put in stage B, and is placed in the queue with priority `veryslow`.

— Execution in stage A uses f_r, afterwards the propagator is put in stage B, and placed in the queue with priority `veryslow`, unless each x_i has a range domain in which case it is removed from all queues (stage NONE).

Table XIV.   Combination experiments.

Example	immediate		multiple		staged	
	time	steps	time	steps	time	steps
`donald-b`	+0.7%	±0.0%	−15.4%	+10.4%	−16.9%	+10.1%
`donald-d`	−1.6%	−2.4%	±0.0%	+111.9%	−1.6%	+88.1%
`golomb-10-b`	+0.3%	±0.0%	−10.1%	+9.2%	−9.8%	+9.2%
`golomb-10-d`	−2.7%	+0.2%	−14.5%	+8.6%	−16.1%	+8.6%
`graph-color`	−0.6%	−0.6%	+2.0%	+23.0%	−14.4%	+12.3%
`o-latin-7-d`	−1.2%	+1.0%	−9.6%	+14.3%	−14.0%	+10.9%
`partition-32`	−0.6%	±0.0%	−4.3%	+7.4%	−6.2%	+4.2%
`photo`	±0.0%	±0.0%	−0.9%	+7.8%	−1.5%	+7.8%
`picture`	+0.3%	±0.0%	−0.1%	±0.0%	−3.4%	±0.0%
`square-5-d`	−29.6%	−7.4%	−38.3%	+28.8%	−42.8%	+56.0%
`square-7-b`	−0.1%	±0.0%	−17.5%	−11.0%	−18.6%	−10.9%
average (above)	−3.6%	−0.9%	−10.7%	+16.1%	−14.0%	+15.3%
average (all)	−1.6%	−0.4%	−4.8%	+6.8%	−6.5%	+6.5%

— Execution in stage B uses f_s, afterwards the propagator is removed from all queues (stage NONE).

The staged propagator is advantageous since the "fast" propagator f_r can more often determine that its result $D' = f_r(D)$ is also a fixpoint for f_s.   □

Staged propagators are widely applicable. They can be used similarly for the bounds($\mathbb{Z}$) version of the `alldifferent` constraint. Another area where staged propagators can be used is constraint-based scheduling, where typically different propagation methods with different strength and efficiency are available [Baptiste et al. 2001].

Staging is not limited to expensive propagators, it is already useful for binary (for example, combining bounds and domain propagation for the absolute value constraint abs(x) = y) and ternary constraints (for example, combining bounds and domain propagation for the multiplication constraint $x \times y = z$).

It is important to note that staging requires a sufficiently rich spectrum of priorities. For example, to use staging for binary or ternary propagators as mentioned above, at least two different priority levels must be available for staging. This explains why the full priority spectrum is useful: here, for binary propagators two priorities `binary-high` and `binary-low` are available. Likewise, `ternary-high` and `ternary-low` are available for ternary propagators.

7.3   Combining Propagation Experiments

Table XIV presents runtime and propagation steps of different propagator combination schemes compared to a propagation engine using the full priority spectrum and all optimizations presented so far. The experiment "immediate" uses a single propagator that always runs the first stage immediately followed by the second stage. For experiment "multiple", multiple propagators for different stages (as discussed in Section 7.1) are used, whereas for experiment "staged" full staging is used (as described in Section 7.2).

A quite surprising result is that "immediate" offers only modest or even no

speedup. The only exception is `square-5-d` using bounds($\mathbb{R}$) propagation immediately before domain propagation for several linear equation propagators.

Using multiple propagators leads to an average reduction in runtime by 10% with a slowdown for just a single example (`graph-color`). As to be expected, the number of propagation steps rises sharply. This is due to the fact that more propagators need to be run and that, similar to the introduction of priorities, propagators with high priority are run more often. The exceptional case of `square-7-b` where the number of propagation steps decreases appears to be a fortunate change in propagation order.

Staged propagation offers another level of improvement over using multiple propagators: all examples now achieve speedup. As fewer propagators must be executed compared to "multiple", also the number of propagation steps decreases (apart from `square-5-d` being another case of changing propagation order). Due to the reduced overhead compared to "multiple", even small examples such as `graph-color` are able to benefit from staged execution.

The exact improvement in runtime of `donald-d` for "immediate" and "staged" is due to early fixpoint detection for a big linear equation propagator as discussed in Example 7.4.

The memory requirements for "immediate" and "staged" are unchanged. The use of multiple propagators for "multiple" leads to an average increase of 6.4% in allocated memory for the examples shown in Table XIV.

In summary, staged propagation is very effective for all examples, so it should clearly be used.

8. EXPERIMENT SUMMARY

In Table XV we summarize the effect of all improvements suggested in this paper. The naive propagation engine is compared to an engine featuring all techniques introduced in this paper: dynamic fixpoint reasoning, $\{\text{dmc}, \text{fix}, \text{bc}\}$ fully dynamic events, dynamic priority based LIFO queuing with the full priority spectrum, and staged propagators.

It is interesting to note that all examples but `warehouse` show an improvement in runtime and that almost 75% of the examples show an improvement of at least 10%. The improvement in runtime does not incur a large increase in memory: the largest increases are for the three `donald-*` problems, where the increase is actually negligible in absolute terms and due to the underlying memory allocation strategy (as discussed in Section 5.2).

The effects of the individual optimizations discussed in this paper could be summarized as follows. Dynamic fixpoint reasoning subsumes static reasoning and is easy to implement, it provides a modest improvement in execution times. Events, while used in all finite domain propagation engines, have less benefit than perhaps was assumed by developers. Using dynamic events again leads to a modest improvement in execution times. The fairness of a FIFO queue strategy is essential for scheduling propagators. While priorities by themselves are not that important they provide a protection against worst case behavior and enable the use of multiple propagators. Staging is an important optimization that can significantly improve performance. All optimizations are available by default in Gecode, the system used

Table XV.   Experiment summary.

Example	no optimizations		all optimizations	
	time (ms)	mem (KB)	time	memory
all-interval-500	118.31	385.4	−36.1%	+25.0%
alpha	106.56	22.2	−31.2%	+9.8%
bibd-7-3-60	2 279.36	6 688.6	−11.0%	+5.8%
cars	4.64	41.7	−7.7%	+1.2%
crowded-chess-7	624.25	196.9	−8.1%	−6.4%
donald-b	0.70	7.4	−25.7%	+19.0%
donald-d	30.37	3.2	−13.0%	+68.2%
donald-v	0.38	5.4	−3.3%	+44.4%
golomb-10-b	1 347.48	40.0	−38.5%	+9.3%
golomb-10-d	2 430.00	37.0	−43.7%	+4.8%
graph-color	35.87	832.4	−19.9%	+9.0%
grocery	55.41	7.7	−5.9%	+7.9%
knights-10	7.46	770.3	−14.7%	+4.2%
minsort-200	342.48	32 454.5	−58.8%	−5.1%
o-latin-7-d	574.36	242.9	−32.5%	+5.5%
partition-32	8 571.24	160.4	−40.4%	+9.2%
photo	108.87	37.0	−23.8%	+10.1%
picture	1 553.42	450.5	−3.0%	+18.0%
queens-400	4 433.12	30 286.1	−87.6%	+0.2%
queens-400-a	16.15	566.3	−10.3%	+4.4%
sequence-500	517.96	6 081.5	−79.8%	−49.4%
square-5-d	33 391.24	43.5	−44.2%	+11.9%
square-7-b	10 166.24	160.3	−41.2%	+8.7%
square-7-v	5 690.00	144.2	−3.9%	+11.7%
warehouse	0.74	29.4	+1.2%	+1.0%
average (all)	—	—	−33.3%	+7.2%

for the examples (see Appendix B).

Scalability. Finally, we examine the issue of how the optimizations scale with problem size. Table XVI shows the results of the benchmarks with scale factors, tried at different sizes. For each problem, the table shows how the number of variables and propagators grows with the increase in problem size n. First one should note that scaling the size of the problem does not necessarily mean that the larger versions of the problem are harder when we are searching for the first solution, although this is usually the case. For example queens-200 is substantially harder than queens-400, square-5-v is substantially harder than square-6-v.

For benchmarks where size does not correspond to difficulty queens and square it is hard to see any pattern. In general in the other results we can see two main behaviors for scaling. The improvements/space penalty from optimization may be more or less constant across size, e.g. time and steps for all-interval and partition, memory for partition and queens. The improvements may increase with size: time for golomb, time and steps for knights, sequence, memory for all-interval, golomb. The only example where we have a trend in the wrong direction is size for knights, and steps for golomb, both of which are do not have a negative effect on time. Overall in almost all examples increasing the size of the

Table XVI. Experiments to demonstrate scalability.

Size	no optimizations			all optimizations		
	time (ms)	steps	mem (KB)	time	steps	memory
all-interval-n: $O(n)$ variables, $O(n)$ propagators						
500	119.12	503 036	385.4	−36.0%	−24.6%	+25.0%
1000	484.56	2 006 041	769.4	−36.9%	−24.8%	+25.0%
1500	1 155.00	4 509 046	1 537.5	−39.1%	−24.9%	+16.7%
2000	2 093.94	8 012 046	2 049.5	−39.8%	−24.9%	+12.5%
golomb-n-b: $O(n^2)$ variables, $O(n^2)$ propagators						
8	23.50	49 602	17.5	−38.1%	+7.9%	+20.6%
9	165.18	330 795	26.8	−37.5%	+21.9%	+13.6%
10	1 346.24	2 642 464	40.0	−38.5%	+30.8%	+9.3%
11	29 241.24	56 282 529	54.2	−39.6%	+39.3%	+8.6%
knights-n: $O(n^2)$ variables, $O(n^2)$ propagators						
10	7.44	48 038	770.3	−14.5%	−57.9%	+4.2%
20	59.87	606 139	3 085.1	−37.9%	−74.3%	+8.3%
30	721.71	7 620 518	18 718.6	−50.5%	−82.7%	+6.8%
40	1 773.12	17 674 449	31 665.9	−63.0%	−89.3%	+7.9%
partition-n: $O(n)$ variables, $O(n)$ propagators						
24	851.84	1 712 925	106.9	−40.6%	−31.0%	+8.9%
28	2 723.12	5 402 047	135.2	−40.6%	−30.8%	+10.9%
32	8 605.60	16 785 121	160.4	−40.3%	−31.5%	+9.2%
36	69 418.12	131 344 889	217.4	−41.1%	−30.0%	+9.8%
queens-n: $O(n)$ variables, $O(n^2)$ propagators						
100	47.51	543 699	1 668.0	−42.9%	−91.7%	±0.0%
200	24 646.84	253 087 619	5 958.2	−26.4%	−56.2%	±0.0%
300	1 583.74	13 345 211	15 049.9	−83.4%	−98.8%	±0.0%
400	4 407.87	31 424 152	30 286.1	−87.8%	−99.1%	+0.2%
queens-n-a: $O(n)$ variables, $O(1)$ propagators						
100	1.37	828	54.0	−3.5%	−41.1%	+12.4%
200	4 324.36	1 769 276	149.0	−0.5%	−21.7%	+20.6%
300	8.64	1 849	306.2	−11.7%	−49.0%	+4.2%
400	16.22	2 469	566.3	−10.2%	−48.8%	+4.4%
sequence-n: $O(n)$ variables, $O(n)$ propagators						
250	74.18	41 786	1 665.5	−71.0%	−43.6%	−49.9%
500	517.50	151 610	6 081.5	−79.7%	−50.1%	−49.4%
750	1 777.65	316 548	13 377.5	−78.6%	−55.1%	−37.3%
1000	4 764.06	564 891	23 297.6	−81.6%	−51.9%	−48.1%
square-n-v: $O(n^2)$ variables, $O(n)$ propagators						
4	10.08	35 150	20.9	−6.9%	−22.8%	+18.2%
5	876.84	2 905 924	43.5	−5.1%	−21.3%	+7.7%
6	0.57	1 655	56.9	−1.8%	−20.1%	+13.0%
7	5 746.24	13 956 982	144.2	−5.6%	−32.3%	+11.7%

problem either keeps the same level of optimizations, or improves it. Hence the optimizations are scalable.

9. CONCLUSION AND FUTURE WORK

We have given a formal definition of propagation systems including idempotence, events, and priorities used in current propagation systems and have evaluated their impact. We have introduced dynamically changing event sets which are shown

to improve efficiency considerably. The paper has introduced multiple and staged propagators which are shown to be an important optimization in particular for improving the efficiency of costly global constraints.

While the improvements to an engine of a propagation based constraint solver have been discussed for integer constraints, the techniques readily carry over to arbitrary constraint domains such as finite sets and multisets.

A rather obvious way to further speed up constraint propagation is to consider not only cost but also estimated impact for a propagator. However, while computing cost is straightforward it is currently not clear to us how to accurately predict propagation impact.

A.   EXAMPLES USED IN EXPERIMENTS

All variants of constraint propagation discussed in the paper are experimentally evaluated. The characteristics of the examples used in evaluation are summarized in Table XVII. The column "variables" gives the number of variables in the example, whereas the column "propagators" shows the number of propagators as implementations of constraints in the example. The column "search" shows which search strategy is used to search for a solution ("first" is simple backtracking search for the first solution, "all" is search for all solutions, "best" is branch-and-bound search for a best solution). The two last columns describe how many failed nodes are explored during search (column "failures") and how many solutions are found (column "solutions").

A -d at the end of the example name means that domain propagation is used for all occurring alldifferent and linear equation constraints. Likewise, -b means that bounds($\mathbb{Z}$) propagation is used for all alldifferent and bounds($\mathbb{R}$) for all linear equation constraints. In contrast, for -v bounds($\mathbb{R}$) propagation is used for all linear constraints, whereas naive propagation (eliminating assigned values as in Example 7.1) is used for alldifferent.

If not otherwise mentioned, bounds($\mathbb{R}$) consistency is used for arithmetic constraints (including linear constraints) and naive propagation for alldifferent.

— all-interval-500 computes a series of numbers where the distances between adjacent numbers are pairwise distinct (prob007 in [CSPLib 2006]). The model uses a single bounds($\mathbb{Z}$) consistent alldifferent propagator and many binary absolute value ($\text{abs}(x) = y$) and ternary minus propagators.

— alpha and donald are crypto-arithmetic puzzles involving linear equation propagators and a single alldifferent propagator.

— bibd-7-3-60 is an instance of a balanced incomplete block design problem with parameters $(v, k, l) = (7, 3, 60)$ (prob028 in [CSPLib 2006]). The model involves Boolean-sum propagators and lex propagators for symmetry breaking.

— cars models the well known car sequencing problem from [Van Hentenryck et al. 1992] using element, exactly, and linear equation propagators (prob001 in [CSPLib 2006]).

— crowded-chess-7 places different chess pieces on a 7×7 chessboard [Dudeney 1958]. It uses exactly, element, domain consistent alldifferent, and bounds($\mathbb{R}$) consistent linear equation propagators.

Table XVII.   Example characteristics.

Example	variables	propagators	search	failures	solutions
`all-interval-500`	1 498	1 002	first	0	1
`alpha`	26	21	all	7 435	1
`bibd-7-3-60`	11 760	9 693	first	1 306	1
`cars`	60	93	all	107	6
`crowded-chess-7`	163	275	first	30 396	1
`donald-b`	10	2	first	79	1
`donald-d`	10	2	first	5	1
`donald-v`	10	2	first	79	1
`golomb-10-b`	46	46	best	19 929	10
`golomb-10-d`	46	46	best	19 929	10
`graph-color`	201	566	first	37	1
`grocery`	7	7	first	37	1
`knights-10`	2 028	2 981	first	2	1
`minsort-200`	399	398	first	0	1
`o-latin-7-d`	147	133	first	2 188	1
`partition-32`	128	134	first	160 258	1
`photo`	61	54	best	6 995	7
`picture`	625	50	first	3 242	1
`queens-400`	400	239 400	first	10	1
`queens-400-a`	400	3	first	10	1
`sequence-500`	500	502	all	250	1
`square-5-d`	25	15	first	41 272	1
`square-7-b`	49	19	first	245 208	1
`square-7-v`	49	19	first	481 301	1
`warehouse`	81	76	best	20	4

— `golomb-10` finds an optimal Golomb ruler of size 10 (`prob006` in [CSPLib 2006]) with the usual model.

— `graph-color` performs clique-based graph coloring for a graph with 200 nodes. Coloring each clique uses a domain consistent `alldifferent` propagator.

— `grocery` is a small crypto-arithmetic puzzle using in particular bounds($\mathbb{R}$) consistent multiplication propagators.

— `knights-10` finds a sequence of knight moves on a 10×10 chess board such that each field is visited exactly once and that the moves return the knight to the starting field. The model uses a naive `alldifferent` propagator and a large number of reified binary propagators.

— `minsort-200` sorts 200 variables using 200 minimum propagators.

— `o-latin-7` finds an orthogonal latin square of size 7 and mostly uses domain consistent `alldifferent` propagators.

— `partition-32` partitions two 32 number blocks such that their products match. It uses several bounds($\mathbb{R}$) multiplication propagators, a single domain consistent `alldifferent` propagator, and few linear equation propagators.

— `photo` places 9 persons on a picture such that as many preferences as possible are satisfied. Uses a large bounds($\mathbb{Z}$) consistent `alldifferent` propagator, a large bounds($\mathbb{R}$) consistent linear propagator, and many reified binary propagators.

— `picture` models a 25×25 picture-puzzle (`prob012` in [CSPLib 2006]) using 50 `regular` propagators.

— `queens-400` and `queens-400-a` place 400 queens on a chess board of size 400×400 such that the queens do not attack each other. `queens-400` uses quadratically many binary disequality propagators, while `queens-400-a` uses three naive `alldifferent`-propagators.

— `sequence-500` finds a magic sequence with 500 elements using 500 `exactly` propagators (`prob019` in [CSPLib 2006]).

— `square-5` (`square-7`) computes a magic square of size 5×5 (7×7) using linear equation propagators and a single `alldifferent` propagator (`prob019` in [CSPLib 2006]).

— `warehouse` solves a warehouse location problem following [Van Hentenryck 1999].

B. EVALUATION PLATFORM

All experiments use Gecode, a C++-based constraint programming library [Gecode Team 2006]. Gecode is one of the fastest constraint programming systems currently available, benchmarks comparing Gecode to other systems are available from Gecode's webpage. The version used in this paper corresponds to Gecode 1.3.0 (albeit slightly modified to ease the numerous experiments in this paper). Gecode has been compiled with Microsoft Visual Studio Express Edition 2005.

All examples have been run on a Laptop with a 2 GHz Pentium M CPU and 1024 MB main memory running Windows XP. Runtimes are the average of 25 runs with a coefficient of deviation less than 4% for all benchmarks.

REFERENCES

APT, K. 2003. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, United Kingdom.

BAPTISTE, P., LE PAPE, C., AND NUIJTEN, W. 2001. *Constraint-based Scheduling.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

BELDICEANU, N., HARVEY, W., HENZ, M., LABURTHE, F., MONFROY, E., MÜLLER, T., PERRON, L., AND SCHULTE, C. 2000. Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. Technical Report TRA9/00, School of Computing, National University of Singapore. Sept.

BENHAMOU, F. 1996. Heterogeneous Constraint Solving. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming.* Lecture Notes in Computer Science, vol. 1139. Springer-Verlag, Aachen, Germany, 62–76.

CARLSSON, M. AND BELDICEANU, N. 2002. Revisiting the lexicographic ordering constraint. Tech. Rep. T2002-17, Swedish Institute of Computer Science, Stockholm, Sweden.

CARLSSON, M., OTTOSSON, G., AND CARLSON, B. 1997. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97.* Lecture Notes in Computer Science, vol. 1292. Springer-Verlag, Southampton, United Kingdom, 191–206.

CHAMARD, A., FISCHLER, A., GUINAUDEAU, D.-B., AND GUILLARD, A. 1995. CHIC lessons on CLP methodology. Tech. rep., Dassault Aviation.

CHOI, C. W., HARVEY, W., LEE, J. H.-M., AND STUCKEY, P. J. 2006. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 4304. Springer-Verlag, Berlin, Germany, 49–58.

CODOGNET, P. AND DIAZ, D. 1996. Compiling constraints in clp(FD). *Journal of Logic Programming 27,* 3 (June), 185–226.

CSPLIB. 2006. CSPLib: a problem library for constraints. Available from http://www.csplib.org.

DUDENEY, H. E. 1958. *Amusements in Mathematics*. Dover, New York, NY, USA.

GECODE TEAM. 2006. Gecode: Generic constraint development environment. Available from http://www.gecode.org.

GENT, I. P., JEFFERSON, C., AND MIGUEL, I. 2006. Watched literals for constraint propagation in Minion. In *Twelfth International Conference on Principles and Practice of Constraint Programming*, F. Benhamou, Ed. Lecture Notes in Computer Science, vol. 4204. Springer-Verlag, Nantes, France, 182–197.

GRANVILLIERS, L. AND MONFROY, E. 2003. Implementing constraint propagation by composition of reductions. In *Logic Programming: Proceedings of the* 19th *International Conference*. Lecture Notes in Computer Science, vol. 2916. Springer-Verlag, Mumbai, India, 300–314.

HARVEY, W. 2004. Personal communication.

HARVEY, W. AND STUCKEY, P. J. 2003. Improving linear constraint propagation by changing constraint representation. *Constraints 8,* 2, 173–207.

ILOG S.A. 2000. *ILOG Solver 5.0: Reference Manual*. Gentilly, France.

INTELLIGENT SYSTEMS LABORATORY. 2004. SICStus Prolog user's manual, 3.11.1. Tech. rep., Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden.

LABURTHE, F. 2000. CHOCO: implementing a CP kernel. See Beldiceanu et al. [2000], 71–85.

LAGERKVIST, M. Z. AND SCHULTE, C. 2007. Advisors for incremental propagation. In *13th International Conference on Principles and Practice of Constraint Programming*, C. Bessière, Ed. Lecture Notes in Computer Science. Springer-Verlag, Providence, RI, USA, 409–422.

LHOMME, O., GOTLIEB, A., AND RUEHER, M. 1998. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming 37,* 1–3, 165–183.

MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence 8,* 1, 99–118.

MARRIOTT, K. AND STUCKEY, P. J. 1998. *Programming with Constraints: an Introduction*. The MIT Press, Cambridge, MA, USA.

MOHR, R. AND HENDERSON, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence 28*, 225–233.

MOHR, R. AND MASINI, G. 1988. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI 88)*, Y. Kodratoff, Ed. Pitmann Publishing, Munich, Germany, 651–656.

MONFROY, E. 1990. Using "weaker" functions for constraint propagation over real numbers. In *Proceedings of The 14th ACM Symposium on Applied Computing*. ACM, San Antonio, TX, USA, 553–559. Scientific Computing Track.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*. ACM, Las Vegas, NV, USA, 530–535.

MOZART CONSORTIUM. 1999. The Mozart programming system. Available from www.mozart-oz.org.

PESANT, G. 2004. A regular language membership constraint for finite sequences of variables. In *10th International Conference on Principles and Practice of Constraint Programming*, M. Wallace, Ed. Lecture Notes in Computer Science, vol. 3258. Springer-Verlag, Toronto, Canada, 482–495.

PUGET, J.-F. 1998. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*. AAAI Press/The MIT Press, Madison, WI, USA, 359–366.

RÉGIN, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Vol. 1. AAAI Press, Seattle, WA, USA, 362–367.

SAVÉANT, P. 2000. Constraint reduction at the type level. See Beldiceanu et al. [2000], 16–29.

SCHULTE, C. AND STUCKEY, P. J. 2004. Speeding up constraint propagation. In *10th International Conference on Principles and Practice of Constraint Programming*, M. Wallace, Ed. Lecture Notes in Computer Science, vol. 3258. Springer-Verlag, Toronto, Canada, 619–633.

SCHULTE, C. AND STUCKEY, P. J. 2005. When do bounds and domain propagation lead to the same search space? *ACM Trans. Program. Lang. Syst. 27,* 3 (May), 388–425.

VAN HENTENRYCK, P. 1999. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, MA, USA.

VAN HENTENRYCK, P., SARASWAT, V., AND DEVILLE, Y. 1991. Constraint processing in cc(FD). Draft.

VAN HENTENRYCK, P., SARASWAT, V., AND DEVILLE, Y. 1998. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming 37,* 1–3, 139–164.

VAN HENTENRYCK, P., SIMONIS, H., AND DINCBAS, M. 1992. Constraint satisfaction using constraint logic programming. *Artificial Intelligence 58,* 113–159.

WALLACE, M., NOVELLO, S., AND SCHIMPF, J. 1997. Eclipse: A platform for constraint logic programming. Tech. rep., IC-Parc, Imperial College, London, GB. Aug.

ZHOU, N.-F. 2006. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming 6,* 5 (Sept.), 483–508.