View-based Propagator Derivation

Christian Schulte
ICT, KTH - Royal Institute of Technology, Sweden, cschulte@kth.se

Guido Tack
Programming Systems Lab, Saarland University, Germany, tack@ps.uni-sb.de

Abstract

When implementing a propagator for a constraint, one must decide about variants: When implementing min, should one also implement max? Should one implement linear constraints both with unit and non-unit coefficients? Constraint variants are ubiquitous: implementing them requires considerable (if not prohibitive) effort and decreases maintainability, but will deliver better performance than resorting to constraint decomposition.

This paper shows how to use views to derive *perfect* propagator variants. A model for views and derived propagators is introduced. Derived propagators are proved to be indeed perfect in that they inherit essential properties such as correctness and domain and bounds consistency. Techniques for systematically deriving propagators such as transformation, generalization, specialization, and type conversion are developed. The paper introduces an implementation architecture for views that is independent of the underlying constraint programming system. A detailed evaluation of views implemented in Gecode shows that derived propagators are efficient and that views often incur no overhead. Without views, Gecode would either require 180 000 rather than 40 000 lines of propagator code, or would lack many efficient propagator variants. Compared to 8 000 lines of code for views, the reduction in code for propagators yields a 1750% return on investment.

1 Introduction

When implementing a propagator for a constraint, one typically must also decide whether to implement some of its variants. When implementing a propagator for the constraint $\max\{x_1,\dots,x_n\} = y$, should one also implement $\min\{x_1,\dots,x_n\} = y$? The latter can be implemented using the former as $\max\{-x_1,\dots,-x_n\} = -y$. When implementing a propagator for the linear equation $\sum_{i=1}^{n} a_i x_i = k$ for integer variables x_i and integers a_i and k, should one also implement the special case $\sum_{i=1}^{n} x_i = k$ for better performance? When implementing a propagator for the reified linear equation $(\sum_{i=1}^{n} x_i = c) \leftrightarrow b$, should one also implement $(\sum_{i=1}^{n} x_i \neq c) \leftrightarrow b$? These two constraints only differ by the sign of b, as the latter is equivalent to $(\sum_{i=1}^{n} x_i = c) \leftrightarrow \neg b$.

The two straightforward approaches for implementing constraint variants are to either implement dedicated propagators for the variants, or to decompose. In the last example, for instance, the reified constraint could be decomposed into two propagators, one for $(\sum_{i=1}^{n} x_i = c) \leftrightarrow b'$, and one for $b \leftrightarrow \neg b'$, introducing an additional variable b'.

Implementing the variants inflates code and documentation and is error prone. Given the potential code explosion, one may be able to only implement some variants (say, min and max). Other variants important for performance (say, ternary min and max) may be infeasible due to excessive programming and maintenance effort. Decomposing, on the other hand, massively increases memory consumption and runtime.

This paper introduces a third approach: *deriving* propagators from already existing propagators using *views*. This approach combines the efficiency of dedicated propagator implementations with the simplicity and effortlessness of decomposition.

Example 1 (Deriving a minimum propagator) Consider a propagator for the constraint $\max(x,y) = z$. Given three additional propagators for $x' = -x$, $y' = -y$, and $z' = -z$, we could propagate the constraint $\min(x',y') = z'$ using the propagator for $\max(x,y) = z$. Instead, this paper proposes to derive a propagator using views that perform the simple transformations corresponding to the three additional propagators.

Views transform input and output of a propagator. For example, a minus view on a variable x transforms the variable domain of x by negating each element, passes the transformed domain to the propagator, and performs the inverse transformation on the domain returned by the propagator. With views, the implementation of the maximum propagator can be reused: a propagator for the minimum constraint can be derived from a propagator for the maximum constraint and a minus view for each variable. $*$

This paper contributes an implementation-independent model for views and derived propagators, techniques for deriving propagators, concrete implementation techniques, and an evaluation that shows that views are widely applicable, drastically reduce programming effort, and yield an efficient implementation.

More specifically, we identify the properties of views that are essential for deriving *perfect* propagators. The paper establishes a formal model that defines a view as a function and a derived propagator as functional composition of views (mapping values to values) with a propagator (mapping domains to domains). This model yields all the desired results: derived propagators are indeed propagators; they faithfully implement the intended constraints; domain consistency carries over to derived propagators; different forms of bounds consistency over integer variables carry over provided that the views satisfy additional yet natural properties.

We introduce techniques for deriving propagators that use views for transformation, generalization, specialization, and type conversion of propagators. We show how to apply these techniques for different variable domains using various views and how views can be used for the derivation of dual scheduling propagators.

We present and evaluate different implementation approaches for views and derived propagators. An implementation using parametric polymorphism (such as templates in C++) is shown to incur no or very low overhead. The architecture is orthogonal to the used constraint programming system and has been fully implemented in Gecode [10]. We analyze how successful the use of derived propagators has been for Gecode.

Plan of the paper. Section 2 introduces constraints and propagators. Section 3 establishes views and propagator derivation. Section 4 presents propagator derivation techniques. Section 5 describes an implementation architecture based on parametric propagators and range iterators. Section 6 discusses limitations of views. The implementation is evaluated in Section 7, and Section 8 concludes.

2 Preliminaries

This section introduces constraints, propagators, and propagation strength.

Variables, constraints, and domains. Constraint satisfaction problems use a *finite set of variables* X and a *finite set of values* V. We typically write variables as $x,y,z \in X$ and values as $v,w \in V$.

A solution of a constraint satisfaction problem assigns a single value to each variable. A constraint restricts which assignments of values to variables are allowed.

Definition 1 (Assignments and constraints) An *assignment* a is a function mapping variables to values. The set of all assignments is $\mathsf{Asn} = X \to V$. A *constraint* c is a set of assignments, $c \in \mathsf{Con} = \mathscr{P}(\mathsf{Asn}) = \mathscr{P}(X \to V)$ (we write $\mathscr{P}(S)$ for the power set of S). Any assignment $a \in c$ is a *solution* of c. $*$

Constraints are defined on assignments as total functions on all variables. For a typical constraint c, only a subset $\mathrm{vars}(c)$ of the variables is *significant*; the constraint is the full relation for all $x \notin \mathrm{vars}(c)$. Constraints are either written as sets of assignments (for example, $\{a \in \mathsf{Asn} \mid a(x) < a(y)\}$) or as expressions with the usual meaning, using the notation $[\![\cdot]\!]$ (for example, $[\![x < y]\!]$).

Example 2 (Sum constraint) Let $X = \{x,y,z\}$ and $V = \{1,2,3,4\}$. The constraint $[\![x = y + z]\!]$ corresponds to the following set of assignments:

$$\begin{aligned}[\![x = y+z]\!] &= \{(x \mapsto a, y \mapsto b, z \mapsto c) \mid a,b,c \in V \wedge a = b + c\} \\ &= \{(x \mapsto 2, y \mapsto 1, z \mapsto 1), (x \mapsto 3, y \mapsto 1, z \mapsto 2), \\ &\quad\ (x \mapsto 3, y \mapsto 2, z \mapsto 1), (x \mapsto 4, y \mapsto 2, z \mapsto 2)\}\end{aligned}$$ $*$

Definition 2 (Domains) A *domain* d is a function mapping variables to sets of values, such that $d(x) \subseteq V$. The set of all domains is $\mathsf{Dom} = X \to \mathscr{P}(V)$. The set of values in d for a particular variable x, $d(x)$, is called the *variable domain* of x. A domain d represents a set of assignments, a constraint, defined as

$$\mathrm{con}(d) = \{a \in \mathsf{Asn} \mid \forall x \in X : a(x) \in d(x)\}$$

An assignment $a \in \mathrm{con}(d)$ is *licensed* by d. $*$

Domains thus represent *Cartesian* sets of assignments. In this sense, any domain is also a constraint. For a more uniform representation, we take the liberty to use domains as constraints. In particular, $a \in d$ (instead of $a \in \mathrm{con}(d)$) denotes an assignment a licensed by d, and $c \cap d$ denotes $c \cap \mathrm{con}(d)$.

A domain d that maps some variable to the empty value set is *failed*, written $d = \emptyset$, as it represents no valid assignments ($\mathrm{con}(d) = \emptyset$). A domain d representing a single assignment, $\mathrm{con}(d) = \{a\}$, is *assigned*, and is written as $d = \{a\}$.

Definition 3 (Constraint satisfaction problems) A *constraint satisfaction problem* (CSP) is a pair $\langle d, C\rangle$ of a domain d and a set of constraints C. The *solutions* of a CSP $\langle d, C\rangle$ are the assignments licensed by d that satisfy all constraints in C, defined as $\mathrm{sol}(\langle d, C\rangle) = \{a \in \mathrm{con}(d) \mid \forall c \in C : a \in c\}$. $*$

Propagators. A propagation-based constraint solver employs *propagators* to implement constraints. A propagator for a constraint c takes a domain d as input and removes values from the variable domains in d that are in conflict with c.

A domain d is *stronger* than a domain d', written $d \subseteq d'$, if and only if $d(x) \subseteq d'(x)$ for all $x \in X$. A domain d is *strictly stronger* than a domain d', written $d \subset d'$, if and only if d is stronger than d' and $d(x) \subset d'(x)$ for some variable x. The goal of constraint propagation is to prune values from variable domains, thus inferring stronger domains, without removing solutions of the constraints.

A propagator is a function p that takes a domain as its argument and returns a stronger domain, it may only *prune* assignments. If the original domain is an assigned domain $\{a\}$, the propagator either accepts ($p(\{a\}) = \{a\}$) or rejects ($p(\{a\}) = \emptyset$) it, realizing a *decision procedure* for its constraint. The pruning and the decision procedure must be consistent: if the decision procedure accepts an assignment, the pruning procedure must never remove this assignment from any domain. This property is enforced by requiring propagators to be monotonic.

Definition 4 (Propagators) A *propagator* is a function $p \in \mathsf{Dom} \to \mathsf{Dom}$ that is

- *contracting:* $p(d) \subseteq d$ for any domain d;
- *monotonic:* $p(d') \subseteq p(d)$ for any domains $d' \subseteq d$.

The set of all propagators is Prop. If a propagator p returns a *strictly* stronger domain ($p(d) \subset d$), we say that *p prunes the domain d*. The propagator p *induces* the unique constraint c_p defined by the set of assignments accepted by p:

$$c_p = \{a \in \mathsf{Asn} \mid p(\{a\}) = \{a\}\} \qquad *$$

Propagators can also be *idempotent* ($p(p(d)) = p(d)$ for any domain d). Idempotency is not required to make propagation sound or complete, but it can make propagation more efficient [33]. Like idempotency, monotonicity as defined here is not necessary for soundness or completeness of a solver [34]. Most definitions and theorems in this paper are independent of whether propagators are monotonic or not. Non-monotonicity will thus only be discussed where it is relevant.

Propagation strength. Each propagator induces a single constraint, but different propagators can induce the same constraint, differing in *strength*. Typical examples are propagators for the *all-different* constraint that perform naive pruning when variables are assigned, or establish bounds consistency [26] or domain consistency [30].

In the literature, propagation strength is usually defined as a property of a domain in relation to a constraint. For example, a domain d is *domain-consistent* (also known as generalized arc-consistent) with respect to a constraint c if $d(x)$ only contains values that appear in at least one solution of c for each variable x. As this paper is concerned with propagators, propagation strength is defined with respect to a propagator.

A propagator p is *domain-complete* if any domain it returns is domain-consistent with respect to c_p. For any constraint c, there is exactly one domain-complete propagator for c (as domains form a lattice). It is defined as $\hat{p}_c(d) = \mathrm{dom}(c_p \cap d)$, where $\mathrm{dom}(c)$ is the *domain relaxation* of c, the strongest domain that contains all assignments of c, $\mathrm{dom}(c) = \min\{d \mid c \subseteq d\}$.

For constraints over integer variables ($V \subseteq \mathbb{Z}$), several weaker notions of propagation strength are known. The most well-known is *bounds consistency*, which in fact can mean one of four special cases: range, bounds(D), bounds($\mathbb{Z}$), and bounds($\mathbb{R}$) consistency (as discussed in [7, 28]).

The first three differ in whether holes are ignored in the original domain, in the resulting domain, or in both, in that order. Holes in a domain are ignored by the function $\mathrm{hull}(d)(x) = [\min(d(x)) \,..\, \max(d(x))]$, which returns the convex hull of a variable domain $d(x)$ in $\mathbb{Z}$. Bounds($\mathbb{R}$) consistency only requires solutions to be found in the real-valued relaxation of the constraint (written $c_{\mathbb{R}}$), and is defined using the real-valued convex hull and domain relaxation (written $\text{hull}_{\mathbb{R}}$ and $\text{dom}_{\mathbb{R}}$). The different notions of bounds consistency give rise to the respective definitions of bounds completeness.

Definition 5 (Bounds completeness) A propagator p is

- range-complete if and only if $p(d) \subseteq \text{dom}(c_p \cap \text{hull}(d))$,
- bounds(D)-complete if and only if $p(d) \subseteq \text{hull}(\text{dom}(c_p \cap d))$,
- bounds($\mathbb{Z}$)-complete if and only if $p(d) \subseteq \text{hull}(\text{dom}(c_p \cap \text{hull}(d)))$, and
- bounds($\mathbb{R}$)-complete if and only if $p(d) \subseteq \text{hull}_{\mathbb{R}}(\text{dom}_{\mathbb{R}}(c_{p\mathbb{R}} \cap \text{hull}_{\mathbb{R}}(d)))$

for any domain d. $*$

3 Views

This section defines views and proves properties of view-derived propagators.

3.1 Views and Derived Propagators

Given a propagator p, a view is represented by two functions, φ and φ^-, that can be composed with p such that $\varphi^- \circ p \circ \varphi$ is the desired derived propagator. The function φ transforms the input domain, and φ^- applies the inverse transformation to the propagator's output domain.

Definition 6 (Variable views and views) A *variable view* $\varphi_x \in V \to V'$ for a variable x is an injective function mapping values to values. The set V' may be different from V, and the corresponding sets of assignments, domains, constraints, and propagators are called Asn', Dom', Con', and Prop', respectively.

Given a family of variable views φ_x for all $x \in X$, we lift them point-wise to assignments: $\varphi_{\mathsf{Asn}}(a)(x) = \varphi_x(a(x))$. A *view* $\varphi \in \mathsf{Con} \to \mathsf{Con}'$ is a family of variable views, lifted to constraints: $\varphi(c) = \{\varphi_{\mathsf{Asn}}(a) \mid a \in c\}$. The *inverse* of a view is defined as $\varphi^-(c) = \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c\}$. $*$

Definition 7 (Derived propagators and constraints) Given a propagator $p \in \mathsf{Prop}'$ and a view φ, the *derived propagator* $\widehat{\varphi}(p) \in \mathsf{Prop}$ is defined as $\widehat{\varphi}(p) = \varphi^- \circ p \circ \varphi$. Similarly, a *derived constraint* is defined to be $\varphi^-(c) \in \mathsf{Con}$ for a given $c \in \mathsf{Con}'$. $*$

Example 3 (Scale views) Given a propagator p for the constraint $c = [\![x = y]\!]$, we want to derive a propagator for $c' = [\![x = 2y]\!]$ using a view φ such that $\varphi^-(c) = c'$.

Intuitively, the function φ leaves x as it is and scales y by 2, while φ^- does the inverse transformation. We thus define $\varphi_x(v) = v$ and $\varphi_y(v) = 2v$. That clarifies the need for different sets V and V', as V' must contain all elements of V multiplied by 2.

The derived propagator is $\widehat{\varphi}(p) = \varphi^- \circ p \circ \varphi$. We say that $\widehat{\varphi}(p)$ "uses a scale view on" y, meaning that φ_y is the function defined as $\varphi_y(v) = 2v$. Similarly, using an identity view on x amounts to φ_x being the identity function on V.

Given the assignment $a = (x \mapsto 2, y \mapsto 1)$, we first apply φ and get $\varphi(\{a\}) = \{(x \mapsto 2, y \mapsto 2)\}$. This is accepted by p and returned unchanged, so φ^- transforms it back to $\{a\}$. Another assignment $a' = (x \mapsto 1, y \mapsto 2)$ is transformed to $\varphi(\{a'\}) = \{(x \mapsto 1, y \mapsto 4)\}$, rejected ($p(\varphi(\{a'\})) = \emptyset$), and the empty domain is mapped to the empty domain by φ^-. The propagator $\widehat{\varphi}(p)$ induces $\varphi^-(c)$. $*$

3.2 Correctness of Derived Propagators

Derived propagators are well-defined and correct: a derived propagator $\widehat{\varphi}(p)$ is in fact a propagator, and it induces the desired constraint ($c_{\widehat{\varphi}(p)} = \varphi^{-}(c_p)$). The proofs of these statements employ the following direct consequences of the definitions of views:

P1. φ and φ^{-} are monotonic by construction (as φ and φ^{-} are defined point-wise).

P2. $\varphi^{-} \circ \varphi = \mathrm{id}$ (the identity function, as φ is injective).

P3. $|\varphi(\{a\})| = 1$, $\varphi(\emptyset) = \emptyset$.

P4. For any view φ and domain $d \in \mathsf{Dom}$, we have $\varphi(d) \in \mathsf{Dom}'$, and for any $d' \in \mathsf{Dom}'$, we have $\varphi^{-}(d') \in \mathsf{Dom}$ (as views are defined point-wise).

Proposition 1 (Correctness) For a propagator p and view φ, $\widehat{\varphi}(p)$ is a propagator. $*$

Proof. The derived propagator is well-defined because both $\varphi(d)$ and $\varphi^{-}(d)$ are domains (see P4 above). We have to show that $\varphi^{-} \circ p \circ \varphi$ is contracting and monotonic.

For contraction, we have $p(\varphi(d)) \subseteq \varphi(d)$ as p is contracting. From monotonicity of φ^{-} (with P1), it follows that $\varphi^{-}(p(\varphi(d))) \subseteq \varphi^{-}(\varphi(d))$. As $\varphi^{-} \circ \varphi = \mathrm{id}$ (with P2), we have $\varphi^{-}(p(\varphi(d))) \subseteq d$, which proves that $\widehat{\varphi}(p)$ is contracting.

Monotonicity is shown as follows for all domains d', d with $d' \subseteq d$:

$$
\begin{array}{rcl}
 & \varphi(d') \subseteq \varphi(d) & (\varphi \text{ monotonic, P1}) \\
\Longrightarrow & p(\varphi(d')) \subseteq p(\varphi(d)) & (p \text{ monotonic}) \\
\Longrightarrow & \varphi^{-}(p(\varphi(d'))) \subseteq \varphi^{-}(p(\varphi(d))) & (\varphi^{-} \text{ monotonic, P1})
\end{array}
$$

In summary, for any propagator p, $\widehat{\varphi}(p) = \varphi^{-} \circ p \circ \varphi$ is a propagator. ∎

Non-monotonic propagators as defined in [34] must at least be *weakly* monotonic, which means that $p(\{a\}) \subseteq p(d)$ for all domains d and assignments $a \in d$. The above proof can be easily adjusted to weakly monotonic propagators by replacing d' with $\{a\}$ and using P3 in the second line of the proof.

Proposition 2 (Induced constraints) Let p be a propagator, and let φ be a view. Then $\widehat{\varphi}(p)$ induces the constraint $\varphi^{-}(c_p)$. $*$

Proof. As p induces c_p, we know $p(\{a\}) = c_p \cap \{a\}$ for all assignments a. With $|\varphi(\{a\})| = 1$ (P3), we have $p(\varphi(\{a\})) = c_p \cap \varphi(\{a\})$. Furthermore, we know that $c_p \cap \varphi(\{a\})$ is either $\emptyset$ or $\varphi(\{a\})$.

- *Case $\emptyset$:* We have $\varphi^{-}(p(\varphi(\{a\}))) = \emptyset = \left\{a' \in \mathsf{Asn} \;\middle|\; a = a' \wedge \varphi_{\mathsf{Asn}}(a) \in c_p\right\} = \varphi^{-}(c_p) \cap \{a\}$.
- *Case $\varphi(\{a\})$:* With P2, we have $\varphi^{-} \circ \varphi = \mathrm{id}$ and hence $\varphi^{-}(p(\varphi(\{a\}))) = \{a\}$. Furthermore, $\varphi^{-}(c_p) \cap \{a\} = \left\{a' \in \mathsf{Asn} \;\middle|\; a = a' \wedge \varphi_{\mathsf{Asn}}(a) \in c_p\right\} = \{a\}$.

Together, this shows that $\varphi^{-} \circ p \circ \varphi(\{a\}) = \{a\} \cap \varphi^{-}(c_p)$. ∎

Another important property is that views preserve contraction: if a propagator p prunes a domain, the pruning will not be lost after transformation by φ^{-}.

Proposition 3 (Views preserve contraction) Let p be a propagator, let φ be a view, and let d be a domain such that $p(\varphi(d)) \subset \varphi(d)$. Then $\widehat{\varphi}(p)(d) \subset d$. $*$

Proof. The definition of $\varphi^-(c)$ is $\{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c\}$. Hence, $|\varphi^-(c)| \leq |c|$. Similarly, we know that $|\varphi(c)| = |c|$. From $p(\varphi(d)) \subset \varphi(d)$, it follows that $|p(\varphi(d))| < |\varphi(d)|$. Together, this yields $|\widehat{\varphi}(p)(d)| < |\varphi(d)| = |d|$. We have seen in Proposition 1 that $\widehat{\varphi}(p)(d) \subseteq d$, so we can conclude that $\widehat{\varphi}(p)(d) \subset d$. ∎

3.3 Completeness of Derived Propagators

Ideally, a propagator derived from a domain- or bounds-complete propagator should inherit its completeness. It turns out to not generally be true for all notions of completeness and all views. This section first shows how bounds($\mathbb{Z}$) completeness is inherited, and then generalizes this result to the other notions.

The key insight is that bounds($\mathbb{Z}$) completeness of propagators derived using a view φ depends on whether φ and φ^- commute with the hull operator, as defined below.

Definition 8 A constraint c is a *φ-constraint* for a view φ if and only if for all $a \in c$, there is a $b \in \mathsf{Asn}$ such that $a = \varphi_{\mathsf{Asn}}(b)$. A view φ is *hull-injective* if and only if $\varphi^-(\mathrm{hull}(\mathrm{dom}(c))) = \mathrm{hull}(\mathrm{dom}(\varphi^-(c)))$ for all φ-constraints c. It is *hull-surjective* if and only if $\varphi(\mathrm{hull}(d)) = \mathrm{hull}(\varphi(d))$ for all domains d. It is *hull-bijective* if and only if it is hull-injective and hull-surjective. $*$

The proofs rely on the additional fact that views commute with set intersection.

Lemma 1 For any view φ, the equation $\varphi^-(c_1 \cap c_2) = \varphi^-(c_1) \cap \varphi^-(c_2)$ holds. $*$

Proof. By definition of φ^-, we have

$$\varphi^-(c_1 \cap c_2) = \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_1 \wedge \varphi_{\mathsf{Asn}}(a) \in c_2\}$$

As φ_{Asn} is a function, this is equal to

$$\{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_1\} \cap \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_2\} = \varphi^-(c_1) \cap \varphi^-(c_2)$$ ∎

Theorem 1 (Bounds($\mathbb{Z}$) **completeness)** Let p be a bounds($\mathbb{Z}$)-complete propagator. For any hull-bijective view φ, the propagator $\widehat{\varphi}(p)$ is bounds($\mathbb{Z}$)-complete. $*$

Proof. From Proposition 2, we know that $\widehat{\varphi}(p)$ induces the constraint $\varphi^-(c_p)$. By monotonicity of φ^- (with P1) and bounds($\mathbb{Z}$) completeness of p, we know that

$$\varphi^- \circ p \circ \varphi(d) \subseteq \varphi^-(\mathrm{hull}(\mathrm{dom}(c_p \cap \mathrm{hull}(\varphi(d)))))$$

We now use the fact that both φ^- and φ commute with hull$(\cdot)$ and set intersection:

$$\begin{aligned}
&\varphi^-(\mathrm{hull}(\mathrm{dom}(c_p \cap \mathrm{hull}(\varphi(d))))) \\
&= \varphi^-(\mathrm{hull}(\mathrm{dom}(c_p \cap \varphi(\mathrm{hull}(d))))) && \text{(hull-surjective)} \\
&= \mathrm{hull}(\mathrm{dom}(\varphi^-(c_p \cap \varphi(\mathrm{hull}(d))))) && \text{(hull-injective)} \\
&= \mathrm{hull}(\mathrm{dom}(\varphi^-(c_p) \cap \varphi^-(\varphi(\mathrm{hull}(d))))) && \text{(commute with } \cap) \\
&= \mathrm{hull}(\mathrm{dom}(\varphi^-(c_p) \cap \mathrm{hull}(d))) && \text{(P2)}
\end{aligned}$$

The second step uses hull injectivity, so it requires $c_p \cap \varphi(\mathrm{hull}(d))$ to be a φ-constraint. All assignments in a φ-constraint have to be the image of some assignment under

φ_{Asn}. This is the case here, as the intersection with $\varphi(\text{hull}(d))$ can only contain such assignments. So in summary, we get

$$\varphi^{-} \circ p \circ \varphi(d) \subseteq \text{hull}(\text{dom}(\varphi^{-}(c_p) \cap \text{hull}(d))$$

which is the definition of $\widehat{\varphi}(p)$ being bounds$(\mathbb{Z})$-complete. ■

Stronger notions of completeness. Similar theorems hold for domain completeness, range and bounds$(\mathbb{Z})$ completeness. The theorems directly follow from the fact that any view φ is *domain-injective*, meaning that $\varphi^{-}(\text{dom}(c)) = \text{dom}(\varphi^{-}(c))$ for all constraints c. We split this statement into the following two lemmas.

Lemma 2 Given a constraint c, let $d = \text{dom}(c)$. Then for all $x \in X$, we have $v \in d(x) \Leftrightarrow \exists a \in c: \ a(x) = v$. $*$

Proof. We prove both directions of the equivalence:

- $\Rightarrow$ There must be such an assignment a because otherwise one can construct a strictly stronger $d' \subset d$ with $v \notin d'(x)$ such that still $c \subseteq d'$.
- $\Leftarrow$ Each domain d' in the intersection $\bigcap\{d' \in \mathsf{Dom} \mid c \subseteq \text{con}(d')\}$ must contain the value $v \in d'(x)$ as $c \subseteq d'$. So for the result of the intersection d, $v \in d(x)$. ■

Lemma 3 Any view φ is domain-injective. $*$

Proof. We have to show that $\varphi^{-}(\text{dom}(c)) = \text{dom}(\varphi^{-}(c))$ holds for any constraint c and any view φ. For clarity, we write the equation including the implicit con$(\cdot)$ operations: $\varphi^{-}(\text{con}(\text{dom}(c))) = \text{con}(\text{dom}(\varphi^{-}(c)))$. By definition of φ^{-} and con$(\cdot)$, we have

$$\begin{aligned}\varphi^{-}(\text{con}(\text{dom}(c))) &= \{a \in \mathsf{Asn} \mid \forall x \in X: \ \varphi_{\mathsf{Asn}}(a)(x) \in \text{dom}(c)(x)\} \\ &= \{a \in \mathsf{Asn} \mid \forall x \in X \ \exists b \in c: \ \varphi_{\mathsf{Asn}}(a)(x) = b(x)\} \quad \text{(Lemma 2)}\end{aligned}$$

As φ_{Asn} is an injective function, we can find such a b that is in the range of φ_{Asn}, if and only if there is also a $b' \in \varphi^{-}(c)$ such that $\varphi_{\mathsf{Asn}}(b') = b$. Therefore, we get

$$\begin{aligned}&\left\{a \in \mathsf{Asn} \,\middle|\, \forall x \in X \ \exists b' \in \varphi^{-}(c): \ a(x) = b'(x)\right\} \\ =&\left\{a \in \mathsf{Asn} \,\middle|\, \forall x \in X: \ a(x) \in \text{dom}(\varphi^{-}(c))(x)\right\} \\ =&\,\text{con}(\text{dom}(\varphi^{-}(c)))\end{aligned}$$

■

The following three theorems express under which conditions the different notions of completeness are preserved when deriving propagators. The proofs for these theorems are analogous to the proof of Theorem 1, using Lemma 3.

Theorem 2 (Bounds(D) **completeness)** Let p be a bounds(D)-complete propagator. For any hull-injective view φ, the propagator $\widehat{\varphi}(p)$ is bounds(D)-complete. $*$

Theorem 3 (Range completeness) Let p be a range-complete propagator. For any hull-surjective view φ, the propagator $\widehat{\varphi}(p)$ is range-complete. $*$

Theorem 4 (Domain completeness) Let p be a domain-complete propagator, and let φ be a view. Then $\widehat{\varphi}(p)$ is domain-complete. $*$

A propagator derived from a bounds($\mathbb{Z}$)-complete propagator and a hull-injective but not hull-surjective view is only bounds($\mathbb{R}$)-complete. This is exactly what we would expect from a propagator for linear equations, as the next example demonstrates.

Example 4 (Linear constraints) A propagator for a linear constraint $c_{\sum} = [\![\sum_{i=1}^{n} x_i = c]\!]$ and n scale views (see Example 3) yield a propagator for a linear constraint with coefficients $c_{\sum a} = [\![\sum_{i=1}^{n} a_i x_i = c]\!]$.

The usual propagator for a linear constraint with coefficients achieves bounds($\mathbb{R}$) consistency in linear time $O(n)$ [15]. However, it *is* bounds($\mathbb{Z}$)-complete for unit coefficients. Hence, the above-mentioned property applies: The propagator for $c_{\sum}$ is bounds($\mathbb{Z}$)-complete, scale views are only hull-injective, so the derived propagator for $c_{\sum a}$ is bounds($\mathbb{R}$)-complete. Implementing the simpler propagator without coefficients and deriving the variant with coefficients yields propagators with exactly the same runtime complexity and propagation strength as manually implemented propagators. $*$

3.4 Additional Properties of Derived Propagators

This section discusses how views can be composed, and how derived propagators behave with respect to idempotency and subsumption.

View composition. A derived propagator permits further derivation. Consider a propagator p and two views φ, φ'. Then $\widehat{\varphi'}(\widehat{\varphi}(p))$ is a perfectly acceptable derived propagator, and properties like correctness and completeness carry over transitively. For instance, we can derive a propagator for $[\![x - y = c]\!]$ from a propagator for $[\![x + y = 0]\!]$, combining an *offset view* ($\varphi_y(v) = v + c$) and a *minus view* ($\varphi'_y(v) = -v$) on y. This yields a propagator for $[\![x + (-(y + c)) = 0]\!] = [\![x - y = c]\!]$.

Fixed points. Schulte and Stuckey [33] show how to optimize the scheduling of propagators that are known to be at a fixed point. Views preserve fixed points of propagators, so the same optimizations apply to derived propagators.

Proposition 4 Let p be a propagator, let φ be a view, and let d be a domain. If $\varphi(d)$ is a fixed point of p, then d is a fixed point of $\widehat{\varphi}(p)$. $*$

Proof. Assume $p(p(\varphi(d))) = p(\varphi(d))$. We have to show $\widehat{\varphi}(p)(d) = \widehat{\varphi}(p)(\widehat{\varphi}(p)(d))$. With the assumption, we can write $\widehat{\varphi}(p)(d) = (\varphi^- \circ p \circ p \circ \varphi)(d)$. We know that $\varphi \circ \varphi^-(c) = c$ if $|\varphi^-(c)| = |c|$. As we first apply φ, this is the case here, so we can add $\varphi \circ \varphi^-$ in the middle, yielding $(\varphi^- \circ p \circ (\varphi \circ \varphi^-) \circ p \circ \varphi)(d)$. With function composition being associative, this is equal to $\widehat{\varphi}(p)(\widehat{\varphi}(p)(d))$. ∎

Subsumption. A propagator is *subsumed* (also known as entailed) by a domain d if and only if for all stronger domains $d' \subseteq d$, $p(d') = d'$. Subsumed propagators cannot do any pruning in the remaining subtree of the search, and can therefore be removed. Deciding subsumption is coNP-complete in general, but for many practically relevant propagators an approximation can be decided easily (such as when a domain becomes assigned). The following theorem states that the approximation is also valid for the derived propagator.

Proposition 5 Let p be a propagator and let φ be a view. The propagator $\widehat{\varphi}(p)$ is subsumed by a domain d if and only if p is subsumed by $\varphi(d)$. $*$

Proof. With P2 we get that $\forall d' \subseteq d : \varphi^-(p(\varphi(d'))) = d'$ is equivalent to $\forall d' \subseteq d : \varphi^-(p(\varphi(d'))) = \varphi^-(\varphi(d'))$. As φ^- is a function, and because it preserves contraction (see Proposition 3), this is equivalent to $\forall d' \subseteq d : p(\varphi(d')) = \varphi(d')$. This can be rewritten to $\forall d'' \subseteq \varphi(d) : p(d'') = d''$ because all $\varphi(d')$ are subsets of $\varphi(d)$. ■

3.5 Related Work

While the idea to systematically derive propagators using views is novel, there are a few related approaches we can point out. Reusing functionality (like a propagator) by wrapping it in an adaptor (like a view) is of course a much more general technique—think of higher-order functions like fold or map in functional programming languages; or chaining command-line tools in Unix operating systems using pipes.

Propagator derivation. Views that perform arithmetic transformations are related to the concept of indexicals (see [5, 36]). An indexical is a propagator that prunes a single variable and is defined in terms of range expressions. In contrast to views, range expressions can involve multiple variables, but on the other hand only operate in one direction. For instance, in an indexical for the constraint $[\![x = y + z]\!]$, the range expression $y + z$ would be used to prune the domain of x, but not for pruning the domains of y or z. Views must work in both directions, which is why they are limited in expressiveness.

Unit propagation in SAT solvers performs propagation for Boolean clauses, which are disjunctions of *literals*, which in turn are positive or negated Boolean variables. In implementations such as MiniSat [9], the Boolean clause propagator is in fact derived from a simple n-ary disjunction propagator and *literal views* of the variables that perform negation for the negative literals.

Constraint composition. Instead of regarding a view φ as *transforming* a constraint c, one can regard φ as *additional* constraints, implementing the decomposition. Assuming $\text{vars}(c) = x_1, \ldots, x_n$, we use additional variables $x'_1, \ldots, x'_n$. Instead of c, we use $c' = c[x_1/x'_1, \ldots, x_n/x'_n]$, which is the same relation as c, but on $x'_1, \ldots, x'_n$. Finally, *n view constraints* $c_{\varphi,i}$ link the original variables to the new variables, each $c_{\varphi,i}$ being equivalent to the relation $x'_i = \varphi_i(x_i)$. The solutions of the decomposition model, restricted to the $x_1, \ldots, x_n$, are exactly the solutions of the original view-based model.

Every view constraint $c_{\varphi,i}$ shares exactly one variable with c and no variable with any other $c_{\varphi,i}$. Thus, the constraint graph is Berge-acyclic [3], and a fixed point can be computed by first propagating all the $c_{\varphi,i}$, then propagating $c[x_1/x'_1, \ldots, x_n/x'n]$, and then again propagating the $c_{\varphi,i}$. This is exactly what $\varphi^- \circ p \circ \varphi$ does. Constraint solvers typically do not provide any means of specifying the propagator scheduling in such a fine-grained way (Lagerkvist and Schulte show how to use propagator groups to achieve this [20]). Thus, deriving propagators using views is also a technique for specifying perfect propagator scheduling.

On a more historical level, a derived propagator is related to the notion of *path consistency*. A domain is path-consistent for a set of constraints, if for any subset $\{x, y, z\}$ of its variables, $v_1 \in d(x)$ and $v_2 \in d(y)$ implies that there is a value $v_3 \in d(z)$ such that the pair (v_1, v_2) satisfies all the (binary) constraints between x and y, the pair (v_1, v_3) satisfies all the (binary) constraints between x and z, and the pair (v_3, v_2) satisfies all the (binary) constraints between z and y [21]. If $\widehat{\varphi}(p)$ is domain-complete for $\varphi^-(c)$, then it achieves path consistency for the constraint $c[x_1/x'_1, \ldots, x_n/x'_n]$ and all the $c_{\varphi,i}$ in the decomposition model.

4 Propagator Derivation Techniques

This section introduces techniques for deriving propagators using views. The techniques capture the transformation, generalization, specialization, and type conversion of propagators and are shown to be widely applicable across variable domains and application areas.

4.1 Transformation

Boolean connectives. For Boolean variables, where $V = \{0,1\}$, the only view apart from identity for Boolean variables captures negation. A *negation view* on x defines $\varphi_x(v) = 1 - v$ for $x \in X$ and $v \in V$. As already noted in Section 3.5, deriving propagators using Boolean views thus means to propagate using *literals* rather than variables.

The obvious application of negation views is to derive propagators for all Boolean connectives from just three propagators. A negation view for x in $x = y$ yields a propagator for $\neg x = y$. From disjunction $x \vee y = z$ one can derive conjunction $x \wedge y = z$ with negation views on x, y, z, and implication $x \rightarrow y = z$ with a negation view on x. From equivalence $x \leftrightarrow y = z$ one can derive exclusive or $x \oplus y = z$ with a negation view on z.

As Boolean constraints are widespread, it pays off to optimize frequently occurring cases of propagators for Boolean connectives. One important propagator is for $\bigvee_{i=1}^{n} x_i = y$ with arbitrarily many variables. Again, conjunction can be derived with negation views on the x_i and on y. Another important propagator implements the constraint $\bigvee_{i=1}^{n} x_i = 1$. A dedicated propagator for this constraint is essential as the constraint occurs frequently and can be implemented efficiently using watched literals, see for example [12]. With views all implementation work is readily reused for conjunction. This shows a general advantage of views: effort put into optimizing a single propagator directly pays off for all other propagators derived from it.

Boolean cardinality. Like the constraint $\bigvee_{i=1}^{n} x_i = 1$, the Boolean cardinality constraint $\sum_{i=1}^{n} x_i \geq c$ occurs frequently and can be implemented efficiently using watched literals (requiring $c+1$ watched literals, Boolean disjunction corresponds to the case where $c = 1$). But also a propagator for $\sum_{i=1}^{n} x_i \leq c$ can be derived using negation views on the x_i with the following transformation:

$$\begin{array}{lclcl} \sum_{i=1}^{n} x_i \leq c & \iff & -\sum_{i=1}^{n} x_i \geq -c & \iff & n - \sum_{i=1}^{n} x_i \geq n - c \\ & \iff & \sum_{i=1}^{n} 1 - x_i \geq n - c & \iff & \sum_{i=1}^{n} \neg x_i \geq n - c \end{array}$$

Reification. Many reified constraints (such as $(\sum_{x=1}^{n} x_i = c) \leftrightarrow b$) also exist in a negated version (such as $(\sum_{x=1}^{n} x_i \neq c) \leftrightarrow b$). Deriving the negated version is trivial by using a negation view on the Boolean control variable b. This contrasts nicely with the effort without views: either the entire code must be duplicated or the parts that perform checking whether the constraint or its negation is subsumed must be factored out and combined differently for the two variants.

Transformation using set views. Set constraints deal with variables whose values are finite sets. Using *complement views* (analogous to Boolean negation, as sets with their usual operations also form a Boolean algebra) on x,y,z with a propagator for $x \cap y = z$ yields a propagator for $x \cup y = z$. A complement view on y yields $x \setminus y = z$.

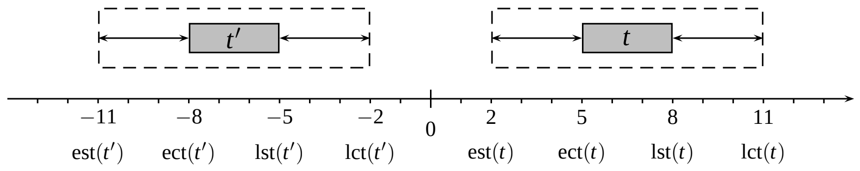

Figure 1: Task t and its dual task t' using a minus view

Transformation using integer views. The obvious integer equivalent to negation views for Boolean variables are *minus views:* a minus view on an integer variable x is defined as $\varphi_x(v) = -v$. Minus views help to derive propagators following simple transformations: for example, $\min(x,y) = z$ can be derived from $\max(x,y) = z$ by using minus views for x, y, and z.

Transformations through minus views can improve performance in subtle ways. Consider a bounds($\mathbb{Z}$)-complete propagator for multiplication $x \times y = z$ (for example, [1, Section 6.5] or [32]). Propagation depends on whether zero is still included in the domains of x, y, or z. Testing for inclusion of zero each time the propagator is executed is inefficient and leads to a convoluted implementation. Instead, one would like to rewrite the propagator to special variants where x, y, and z are either strictly positive or negative. These variants can propagate more efficiently, in particular because propagation can easily be made idempotent. Instead of implementing three different propagators (x,y,z strictly positive; only x or y strictly positive; only z strictly positive), a single propagator assuming that all views are strictly positive is sufficient. The other propagators can be derived using minus views.

Again, with views it becomes realistic to optimize a single implementation of a propagator and derive other, equally optimized, implementations. The effort to implement all required specialized versions without views is typically unrealistic.

Scheduling propagators. An important application area is constraint-based scheduling, see for example [2]. Many propagation algorithms for constraint-based scheduling are based on tasks, where a task t is characterized by its start time, processing time (how long does the task take to be executed on a resource), and end time. Scheduling algorithms are typically expressed in terms of earliest start time ($\mathrm{est}(t)$), latest start time ($\mathrm{lst}(t)$), earliest completion time ($\mathrm{ect}(t)$), and latest completion time ($\mathrm{lct}(t)$).

Another particular aspect of scheduling algorithms is that they are often required in two, mutually dual, variants. Let us consider not-first/not-last propagation as an example. Assume a set of tasks T and a task $t \not\in T$ to be scheduled on the same resource. Then t cannot be scheduled before the tasks in T (t is not-first in $T \cup \{t\}$), if $\mathrm{ect}(t) > \mathrm{lst}(T)$ (where $\mathrm{lst}(T)$ is a conservative estimate of the latest start time of all tasks in T). Hence, $\mathrm{est}(t)$ can be adjusted to leave some room for at least one task from T. The dual variant is that t is not-last: if $\mathrm{ect}(T) > \mathrm{lst}(t)$ (again, $\mathrm{ect}(T)$ estimates the earliest completion time of T), then $\mathrm{lct}(t)$ can be adjusted.

Running the dual variant of a scheduling algorithm on tasks $t \in T$ is the same as running the original algorithm on the *dual tasks* $t' \in T'$, which are simply mirrored at the 0-origin of the time scale (see Figure 1):

$$\mathrm{est}(t') = -\mathrm{lct}(t) \quad \mathrm{ect}(t') = -\mathrm{lst}(t) \quad \mathrm{lst}(t') = -\mathrm{ect}(t) \quad \mathrm{lct}(t') = -\mathrm{est}(t)$$

The dual variant of a scheduling propagator can be automatically derived using a minus view that transforms the time values. In our example, only a propagator for not-first needs to be implemented and the propagator for not-last can be derived (or vice versa). This is in particular beneficial if the algorithms use sophisticated data structures such as Ω-trees [37]. Here, also the data structure needs to be implemented only once and the dual data structure for the dual propagator is derived.

4.2 Generalization

Common views for integer variables capture linear transformations of the integer values: an *offset view* for $o \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = v + o$, and a *scale view* for $a \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = av$.

Offset and scale views are useful for generalizing propagators. Generalization has two key advantages: simplicity and efficiency. A more specialized propagator is often simpler to implement (and simpler to implement correctly) than a generalized version. The specialized version can save memory and runtime during execution.

We can devise an efficient propagation algorithm for a linear equality constraint $\sum_{i=1}^{n} x_i = c$ for the common case that the linear equation has only unit coefficients. The more general case $\sum_{i=1}^{n} a_i x_i = c$ can be derived by using scale views for a_i on x_i (the same technique of course applies to linear inequalities and disequality rather than equality). Similarly, a propagator for *all-different*$(x_1, \ldots, x_n)$ can be generalized to *all-different*$(c_1 + x_1, \ldots, c_n + x_n)$ by using offset views for $c_i \in \mathbb{Z}$ on x_i. Likewise, from a propagator for the element constraint $a[x] = y$ for integers $a_1, \ldots, a_n$ and integer variables x and y, we can derive the generalized version $a[x+o] = y$ with an offset view, where $o \in \mathbb{Z}$ provides a useful offset for the index variable x.

These generalizations can be applied to domain- as well as bounds-complete propagators. While most Boolean propagators are domain-complete, bounds completeness plays an important role for integer propagators. Section 3.3 shows that, given appropriate hull-surjective and/or hull-injective views, the different notions of bounds consistency are preserved when deriving propagators.

The views for integer variables presented in this section have the following properties: minus and offset views are hull-bijective, whereas a scale view for $a \in \mathbb{Z}$ on x is always hull-injective and only hull-surjective if $a = 1$ or $a = -1$ (in which cases it coincides with the identity view or a minus view, respectively).

4.3 Specialization

We employ *constant views* to specialize propagators. A constant view behaves like an assigned variable. In practice, specialization has two advantages. Fewer variables require less memory. And specialized propagators can be compiled to more efficient code, if the constants are known at compile time.

Examples for specialization are

- a propagator for binary linear inequality $x + y \leq c$ derived from a propagator for $x + y + z \leq c$ by using a constant 0 for z;
- a reified propagator for $(x = c) \leftrightarrow b$ from $(x = y) \leftrightarrow b$ and a constant c for y;
- propagators for the counting constraints $|\{i \mid x_i = c\}| = z$ and $|\{i \mid x_i = y\}| = c$ from a propagator for $|\{i \mid x_i = y\}| = z$;
- a propagator for set disjointness from a propagator for $x \cap y = z$ and a constant empty set for z; and many more.

We have to straightforwardly extend the model for constant views. Propagators may now be defined with respect to a superset of the variables, $X' \supseteq X$. A constant view for the value k on a variable $z \in X' \setminus X$ translates between the two sets of variables:

$$\varphi(c) = \{a[k/z] \mid a \in c\} \qquad \varphi^{-}(c) = \left\{a_{|X} \;\middle|\; a \in c\right\}$$

Here, $a[k/z]$ means augmenting the assignment a so that it maps z to k, and $a_{|X}$ is the functional restriction of a to the set X.

It is important that this definition preserves failure. If a propagator returns a failed domain d that maps z to the empty set, then $\varphi^{-}(d)$ is the empty set, too (recall that this is really $\varphi^{-}(\text{con}(d))$, and $\text{con}(d) = \emptyset$ if $d(z) = \emptyset$).

4.4 Type Conversion

A type conversion view lets propagators for one type of variable work with a different type, by translating the underlying representation. Our model already accommodates for this, as a view φ_x maps elements between different sets V and V'.

Integer views. Boolean variables are essentially integer variables restricted to the values $\{0,1\}$. Constraint programming systems may choose a more efficient implementation for Boolean variables and hence the variable types for integer and Boolean variables differ. By wrapping an efficient Boolean variable in an *integer view*, all integer propagators can be directly reused with Boolean variables. This can save substantial effort: for example, an implementation of the *regular*-constraint for Boolean variables can be derived which is actually useful in practice [19].

Singleton set views. A *singleton set view* on an integer variable x, defined as $\varphi_x(v) = \{v\}$, presents an integer variable as a set variable. Many constraints involve both integer and set variables, and some of them can be expressed with singleton set views. A simple constraint is $x \in y$, where x is an integer variable and y a set variable. Singleton set views derive it as $\{x\} \subseteq y$. This extends to $\{x\} \diamond y$ for all other set relations $\diamond$.

Singleton set views can also be used to derive pure integer constraints from set propagators. For example, the constraint $same(x_1,\ldots,x_n,y_1,\ldots,y_m)$ with integer variables x_i, y_i states that the variables x_i take the same values as the variables y_i. With singleton set views, $\bigcup_{i=1}^{n}\{x_i\} = \bigcup_{j=1}^{m}\{y_j\}$ implements this constraint (albeit with weaker propagation than the algorithm presented in [4]).

Set bounds and complete set domain variables. Most systems approximate set variable domains as set intervals defined by lower and upper bounds [25, 13]. However, [16] introduces a representation for the complete domains of set variables, using ROBDDs. Type conversion views can translate between set interval and ROBDD-based implementations. We can derive a propagator on ROBDD-based variables from a set interval propagator, and thus reuse set interval propagators for which no efficient ROBDD representation exists.

4.5 Applicability and Return on Investment

To get an overview of how applicable the presented techniques for propagator derivation are, let us consider the use of views in Gecode (version 3.1.0). Table 1 shows the number of propagator implementations and the number of propagators derived from the

Table 1: Number of implemented vs. derived propagators

Variable type	*Implemented*	*Derived*	*Ratio*
Integer	77	400	5.19
Boolean	28	91	3.25
Set	28	122	4.36
Overall	133	613	4.61

```
class IntVar {
private: int _min, _max;
public:  int min(void) { return _min; }
         int max(void) { return _max; }
         void adjmin(int n) { if (n > _min) _min = n; }
         void adjmax(int n) { if (n < _max) _max = n; }
};

class OffsetView {
protected: IntVar* x; int o;
public:    OffsetView(IntVar* x0, int o0) : x(x0), o(o0) {}
           int min(void) { return x->min()+o; }
           int max(void) { return x->max()+o; }
           void adjmin(int n) { x->adjmin(n-o); }
           void adjmax(int n) { x->adjmax(n-o); }
};
```

Figure 2: Integer variable and offset view

implementations. On average, every propagator implementation results in 4.6 derived propagators. Propagator implementations in Gecode account for more than 40 000 lines of code and documentation. As a rough estimate, deriving propagators using views thus saves around 140 000 lines of code and documentation to be written, tested, and maintained. On the other hand, the views mentioned in this section are implemented in less than 8 000 lines of code, yielding a 1750% return on investment.

5 Implementation

This section presents an implementation architecture for views and derived propagators, based on making propagators *parametric*. Deriving a propagator then means *instantiating* a parametric propagator with views. The presented architecture is an orthogonal layer of abstraction on top of any solver implementation.

5.1 Views

The model introduced views as functions that transform the input and output of a propagator, which maps domains to domains. In an object-oriented implementation of this model, a propagator is no longer a function, but an object with a `propagate` method that *accesses* and *modifies* a domain through the methods of variable objects. Such an object-oriented model is used for example by ILOG Solver [27] and Choco [18], and is the basis of most of the current propagation-based constraint solvers.

Figure 2 shows C++ classes for a simple integer variable (just representing bounds information) and a corresponding offset view. The view has the same interface as the variable, so that it can be used in its place. It contains a pointer to the underlying integer variable and *delegates* all the operations, performing the necessary transformations.

```
template<class VX, class VY>
class Eq : public Propagator {
protected: VX* x; VY* y;
public:    Eq(VX* x0, VY* y0) : x(x0), y(y0) {}
           virtual void propagate(void) {
             x->adjmin(y->min()); x->adjmax(y->max());
             y->adjmin(x->min()); y->adjmax(x->max());
           }
};
```

Figure 3: Parametric equality propagator

5.2 Deriving Propagators

In order to derive a propagator using view objects like the above, we use *parametricity*, a mechanism provided by the implementation language that supports the instantiation of the same code (the propagator) with different parameters (the views).

Figure 3 shows a simple equality propagator. The propagator is based on C++ templates, it is *parametric* over the types of the two views it uses and can be *instantiated* with any view that provides the necessary operations. This type of parametricity is called *parametric polymorphism*, and is available in other programming languages for example in the form of Java generics [14] or Standard ML functors [22].

Given two pointers to integer variables `x` and `y`, the propagator can be instantiated to implement $[\![x = y]\!]$ as follows (using the `IntVar` class from Figure 2):

```
    new Eq<IntVar,IntVar>(x,y);
```

The following instantiation yields a propagator for $[\![x = y + 2]\!]$:

```
    new Eq<IntVar,OffsetView>(x,new OffsetView(y,2));
```

Events. Most constraint solvers schedule the execution of propagators according to *events* (see for example [31]). For example, a propagator p for $[\![x < y]\!]$ can only prune the domain (and thus should only be executed) if either the lower bound of x or the upper bound of y changes, written $\mathsf{lbc}(x)$ and $\mathsf{ubc}(y)$. We say that p *subscribes* to the *event set* $\{\mathsf{lbc}(x), \mathsf{ubc}(y)\}$.

Now assume that p' is derived from p using minus views on x and y, thus implementing $x > y$. Obviously, p' should subscribe to the dual event set, $\{\mathsf{ubc}(x), \mathsf{lbc}(y)\}$. In the implementation, views provide all the operations needed for event handling (such as subscription) and perform the necessary transformations of event sets.

5.3 Parametricity

Independent of the concrete implementation, views form an orthogonal layer of abstraction on top of any propagation-based constraint solver. As long as the implementation language provides some kind of parametricity, and variable domains are accessed through some form of variable objects, propagators can be derived using views.

In addition to parametric polymorphism, two other forms of parametricity exist, *functional* parametricity and *dynamic binding*. Functional parametricity means that in languages such as Standard ML [22] or Haskell [24], a higher-order function is parametric over its arguments. Dynamic binding is typically coupled with inheritance in object-oriented languages (virtual function calls in C++, method calls in Java). Even in languages that lack direct support for parametricity, parametric behavior can often be achieved using other mechanisms, such as macros or function pointers in C.

Choice of parametricity. In C++, parametric polymorphism and dynamic binding have advantages and disadvantages as it comes to deriving propagators.

Templates are compiled by *monomorphization:* the code is replicated and specialized for each instance. The compiler can generate optimized code for each instance, for example by inlining the transformations that a view implements.

Achieving high efficiency in C++ with templates sacrifices some expressiveness. Instantiation can *only* happen at compile-time. Hence, either C++ must be used for modeling, or all potentially required propagator variants must be instantiated explicitly. The *choice* which propagator to use can however be made at runtime: for linear equations, for instance, if all coefficients are units, the optimized original propagator can be posted.

For *n*-ary constraints, compile-time instantiation can be a limitation, as all arrays must be monomorphic (contain only a single kind of view). For example, one cannot mix scale and minus views in linear constraints. For some propagators, we can work around this restriction using more than a single array of views. For example, a propagator for a linear constraint can employ two arrays of different view types, one of which may then be instantiated with identity views and the other with minus views. While this poses a limitation in principle, our experience from Gecode suggests that there are only few propagators in practice that suffer from this limitation.

Dynamic binding is more flexible than parametric polymorphism, as instantiation happens at runtime and arrays can be polymorphic. This flexibility comes at the cost of reduced efficiency, as the transformations done by view operations typically cannot be inlined and optimized, but require additional virtual method calls. Section 7 evaluates empirically how these virtual method calls affect performance.

Compile-time versus runtime constants. Some views involve a parameter, such as the coefficient of a scale view or the constant of a constant view. These parameters can again be instantiated at compile-time or at runtime. For instance, one can regard a minus view as a compile-time specialization of a scale view with coefficient -1, and a zero view may specialize a constant view. With the constants being known at compile-time, the compiler can apply more aggressive optimizations.

5.4 Iterators

Typical domain operations involve a single integer value, for instance adjusting the minimum or maximum of an integer variable. These operations are not efficient if a propagator performs full domain reasoning on integer variables or deals with set variables. Therefore, set-valued operations, like updating a whole integer variable domain to a new set, or excluding a set of elements from a set variable domain, are important for efficiency. Many constraint programming systems provide an abstract set-datatype for accessing and updating variable domains, as for example in Choco [6], ECL^{i}PS^e [8], SICStus Prolog [35], and Mozart [23]. ILOG Solver [17] only allows access by iterating over the values of a variable domain.

This section develops *iterators* as one particular abstract datatype for set-valued operations on variables and views. There are two main reasons to discuss iterators in this paper. First, iterators provide simple, expressive, and efficient set-valued operations on variables. Second, and more importantly, iterators transparently perform the transformations needed for set-valued operations on views, and thus constitute a perfect fit for deriving propagators.

Range sequences and range iterators. A *range* $[m\,..\,n]$ denotes the set of integers $\{l \in \mathbb{Z} \mid m \leq l \leq n\}$. A *range sequence* $\text{ranges}(S)$ for a finite set of integers $S \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle [m_1\,..\,n_1], \ldots, [m_k\,..\,n_k] \rangle$ such that $S = \bigcup_{i=1}^{k} [m_i\,..\,n_i]$ and the ranges are ordered by their smallest elements ($m_i \leq m_{i+1}$ for $1 \leq i < k$). We thus define the set covered by a range sequence as $\text{set}(s) = \bigcup_{i=1}^{k} [m_i\,..\,n_i]$. The above range sequence is also written as $\langle [m_i\,..\,n_i] \rangle_{i=1}^{k}$. Clearly, the range sequence of a set is unique, none of its ranges is empty, and $n_i + 1 < m_{i+1}$ for $1 \leq i < k$.

A *range iterator* for a range sequence $s = \langle [n_i\,..\,m_i] \rangle_{i=1}^{k}$ is an object that provides iteration over s: each of the $[m_i\,..\,n_i]$ can be obtained in sequential order but only one at a time. A range iterator r provides the following operations: $r.\texttt{done}()$ tests whether all ranges have been iterated, $r.\texttt{next}()$ moves to the next range, and $r.\texttt{min}()$ and $r.\texttt{max}()$ return the minimum and maximum value for the current range. By $\text{set}(r)$ we refer to the set defined by an iterator r (which must coincide with $\text{set}(s)$).

A range iterator naturally hides its implementation. It can iterate a sequence (for instance an array) directly by position, but it can just as well traverse a linked list or the leaves of a balanced tree, or for example iterate over the union of two other iterators.

Iterators are consumed by iteration. Hence, if the same sequence needs to be iterated twice, a fresh iterator is needed. If iteration is cheap, an iterator can support multiple iterations by providing a reset operation. Otherwise, a *cache iterator* takes an arbitrary range iterator as input, iterates it completely, and stores the obtained ranges in an array. Its operations then use the array. The cache iterator implements a reset operation, so that the possibly costly input iterator is used only once, while the cache iterator can be used as often as needed.

Iterators for variables. The two basic set-valued operations on integer variables are domain access and domain update. For an integer variable x, the operation $x.\texttt{getdom}()$ returns a range iterator for $\text{ranges}(d(x))$. The operation $x.\texttt{setdom}(r)$ updates the variable domain of x to $\text{set}(r)$ given a range iterator r, provided that $\text{set}(r) \subseteq d(x)$. The responsibility for ensuring that $\text{set}(r) \subseteq d(x)$ is left to the programmer.

In order to provide safer and richer operations, we can use *iterator combinators*. For example, an *intersection iterator* $r = \text{iinter}(r_1, r_2)$ combines two range iterators r_1 and r_2 such that $\text{set}(r) = \text{set}(r_1) \cap \text{set}(r_2)$. Similarly, a *difference iterator* $r = \text{iminus}(r_1, r_2)$ yields $\text{set}(r) = \text{set}(r_1) \setminus \text{set}(r_2)$.

Richer set-valued operations are then effortless. The operation $x.\texttt{adjdom}(r)$ adjusts the domain $d(x)$ by $\text{set}(r)$, yielding $d(x) \cap \text{set}(r)$, whereas $x.\texttt{excdom}(r)$ excludes $\text{set}(r)$ from $d(x)$, yielding $d(x) \setminus \text{set}(r)$:

$$x.\texttt{adjdom}(r) = x.\texttt{setdom}(\text{iinter}(x.\texttt{getdom}(), r))$$
$$x.\texttt{excdom}(r) = x.\texttt{setdom}(\text{iminus}(x.\texttt{getdom}(), r))$$

In contrast to the $x.\texttt{setdom}(\cdot)$ operation, the richer set-valued operations are inherently contracting, and thus safer to use when implementing a propagator.

Iterators also serve as the natural interface for operations on set variables, which are usually approximated as set intervals defined by a lower and an upper bound [25, 13]:

$$d(x) = [\text{glb}(d(x))\,..\,\text{lub}(d(x))] = \{s \mid \text{glb}(d(x)) \subseteq s, s \subseteq \text{lub}(d(x))\}$$

In order to access and update these set bounds, propagators use set-valued operations based on iterators: $x.\mathtt{glb}()$ returns a range iterator for $\mathrm{ranges}(\mathrm{glb}(d(x)))$, $x.\mathtt{lub}()$ returns a range iterator for $\mathrm{ranges}(\mathrm{lub}(d(x)))$, $x.\mathtt{adjglb}(r)$ updates the domain of x to $[\mathrm{glb}(d(x)) \cup \mathrm{set}(r), \mathrm{lub}(d(x))]$, and $x.\mathtt{adjlub}(r)$ updates the domain of x to $[\mathrm{glb}(d(x)), \mathrm{lub}(d(x)) \cap \mathrm{set}(r)]$.

Iterator combinators provide the operations that set propagators need: union, intersection, difference, and complement. Many propagators can thus be implemented directly using iterators and do not require any temporary data structures for storing set-valued intermediate results.

All set-valued operations are parametric with respect to the iterator r: any range iterator can be used. As for parametric propagators, an implementor has to decide on the kind of parametricity to use. Gecode uses template-based parametric polymorphism, with the performance benefits due to monomorphization and consequent code optimization mentioned previously.

Advantages. Range iterators provide essential advantages over an explicit set representation. First, any range iterator regardless of its implementation can be used in domain operations. This turns out to result in simple, efficient, and expressive domain updates. Second, no costly memory management is required to maintain a range iterator as it provides access to only one range at a time. Third, the abstractness of range iterators makes them compatible with views and derived propagators: the necessary view transformations can be encapsulated in an iterator, as discussed below.

Iterators for views. As iterators hide their implementation, they are perfectly suited for implementing the transformations required for set-valued operations on views.

Set-valued operations for constant integer views are straightforward. For a constant view v on constant k, the operation $v.\mathtt{getdom}()$ returns an iterator for the singleton range sequence $\langle [k\,..\,k] \rangle$. The operation $v.\mathtt{setdom}(r)$ just checks whether the range sequence of r is empty (in order to detect failure).

Set-valued operations for an offset view are provided by an *offset iterator*. For a range sequence $r = \langle [m_i\,..\,n_i] \rangle_{i=1}^{k}$ and offset c, $\mathrm{ioffset}(r, c)$ iterates $\langle [m_i + c\,..\,n_i + c] \rangle_{i=1}^{k}$. An offset view on x with offset c then implements $\mathtt{getdom}$ as $\mathrm{ioffset}(x.\mathtt{getdom}(), c)$ and $\mathtt{setdom}(r)$ as $x.\mathtt{setdom}(\mathrm{ioffset}(r, -c))$.

For minus views we just give the range sequence, iteration is obvious. For a given range sequence $\langle [m_i\,..\,n_i] \rangle_{i=1}^{k}$, the negative sequence is obtained by reversal and sign change as $\langle [-n_{k-i+1}\,..\,-m_{k-i+1}] \rangle_{i=1}^{k}$. The same iterator for this sequence can be used both for $\mathtt{setdom}$ and $\mathtt{getdom}$ operations. Note that implementing the iterator is involved as it changes direction of the range sequence. There are two different options for changing direction: either the set-valued operations accept iterators in both directions or a cache iterator is used to reverse the direction. Gecode uses the latter and Section 7.2 evaluates the overhead introduced by cache iterators.

A scale iterator provides the necessary transformations for scale views. Assume a scale view on a variable x with a coefficient $a > 0$, and let $\langle [m_i\,..\,n_i] \rangle_{i=1}^{k}$ be a range sequence for $d(x)$. If $a = 1$, the scale iterator does not change the range sequence. Otherwise, the corresponding scaled range sequence is $\langle \{a \times m_1\}, \{a \times (m_1 + 1)\}, \ldots, \{a \times n_1\}, \ldots, \{a \times m_k\}, \{a \times (m_k + 1)\}, \ldots, \{a \times n_k\} \rangle$. For the other direction, assume we want to update the domain using a set S through a scale view. Assume that $\langle [m_i\,..\,n_i] \rangle_{i=1}^{k}$ is a range sequence for S. Then for $1 \leq i \leq k$ the ranges $[\lceil m_i / a \rceil\,..\,\lfloor n_i / a \rfloor]$ correspond to the required variable domain for x, however they do not necessarily form a range sequence as the ranges might be empty, overlapping, or adjacent. Iterating the range sequence is however simple by skipping empty ranges and merging overlapping or adjacent ranges. Scale views for a variable x and a coefficient a in Gecode are restricted to be strictly positive so as to not change the direction of the scaled range sequence. A negative coefficient can be obtained by using a scale view together with a minus view.

A complement view of a set variable x uses a *complement iterator*, which given a range iterator r iterates over $\overline{\text{set}(r)}$.

6 Limitations

Although views are widely applicable, they are no silver bullet. This section explores some limitations of the presented model.

Beyond injective views. Views are required to be injective, as otherwise $\varphi^{-} \circ \varphi$ is no longer the identity function, and derived propagators would not necessarily be contracting. An example for this behavior is a view for the absolute value of an integer variable. Assuming a variable domain $d(x) = \{1\}$, an absolute value view φ would leave the domain as it is, $\varphi(d)(x) = \{1\}$, but the inverse would "invent" the negative value, $\varphi^{-}(\varphi(d))(x) = \{-1, 1\}$. With an adapted definition of derived propagators, such as $\widehat{\varphi}(p)(d) = \varphi^{-}(p(\varphi(d))) \cap d$, non-injective views could be used – however, many of the proofs in this paper rely on injectivity (though some of the theorems possibly still hold for non-injective views).

Multi-variable views. Some multi-variable views that seem interesting for practical applications do not preserve contraction, for instance a view on the sum or product of two variables. The reason is that removing a value through the view would have to result in removing a *tuple* of values from the domain. As domains can only represent Cartesian products, this is not possible in general. Such a view would have two main disadvantages. First, if propagation of the original constraint is strong but does not lead to an actual domain pruning through the views, then the potentially high computational cost for the pruning does not pay off. A cheaper but weaker, dedicated propagation algorithm or a different modeling with stronger pruning is then a better choice. Second, if views do not preserve contraction, then Proposition 5 does not hold. That means that a propagator p cannot easily detect subsumption any longer, as it would have to detect it for $\widehat{\varphi}(p)$ instead of just for itself, p. Systems such as Gecode that disable subsumed propagators (as described in [33]) then lose this potential for optimization.

For contraction-preserving views on multiple variables, all the theorems still hold. Two such views we could identify are a set view of Boolean variables $[b_1, \ldots, b_n]$, behaving like $\{i \mid b_i = 1\}$; and an integer view of Boolean variables $[b_1, \ldots, b_n]$, where b_i is 1 if and only if the integer has value i; as well as the inverse views of these two.

Propagator invariants. Propagators typically rely on certain invariants of a variable domain implementation. If idempotency or completeness of a propagator depend on these invariants, type conversion views lead to problems, as the actual variable implementation behind the view may not respect the same invariants.

For example, a propagator for set variables based on the set interval approximation can assume that adjusting the lower bound of a variable does not affect its upper bound. If this propagator is instantiated with a type conversion view for an ROBDD-based set

Table 2: Results for Gecode 3.1.0, the baseline for the experiments

Benchmark	*time (ms)*	*mem. (KByte)*	*failures*	*propagations*
All-Interval (50)	183.21	148	0	6 685
All-Interval (100)	3 904.21	516	0	25 866
Alpha (naive)	100.00	23	7 435	136 179
BIBD (7-3-60)	1 762.85	4 516	1 306	921 686
Eq-20	1.52	14	54	3 460
Golomb Rulers (Bnd, 10)	423.39	67	8 890	1 181 704
Golomb Rulers (Dom, 10)	607.86	419	8 890	1 181 770
Graph Coloring	324.46	3 910	1 100	125 264
Magic Sequence (Smart, 500)	251.50	4 484	251	84 302
Magic Sequence (GCC, 500)	305.15	330	251	3 908
Partition (32)	5 928.04	265	160 258	12 107 504
Perfect Square	185.54	3 972	150	305 391
Queens (10)	36.88	27	4 992	43 448
Queens (Dom, 10)	103.38	99	3 940	59 508
Queens (100)	1.54	235	22	455
Queens (Dom, 100)	31.83	2 056	8	693
Sorting (400)	1 400.01	151 413	0	459 501
Social Golfers (8-4-9)	193.37	10 254	32	181 290
Social Golfers (5-3-7)	1 199.51	2 117	1 174	852 391
Hamming Codes (20-3-32)	1 140.98	24 746	2 296	753 751
Steiner Triples (9)	120.11	901	1 067	297 501
Sudoku (Set, 1)	3.48	83	0	1 820
Sudoku (Set, 4)	7.30	148	1	3 752
Sudoku (Set, 5)	55.14	514	25	28 038

variable (see Section 4.4), this invariant is violated: if, for instance, the current domain is $\{\{1,2\},\{3\}\}$, and 1 is added to the lower bound, then 3 is removed from the upper bound (in addition to 2 being added to the lower bound). If a propagator reports that it has computed a fixed point based on the assumption that the upper bound cannot have changed, it may actually not be at a fixed point. This potentially results in incorrect propagation, for instance if the propagator could detect failure if it were run again.

7 Evaluation

While Section 3 proved that derived propagators are perfect with respect to the mathematical model, this section shows that in most cases one can also obtain perfect implementations of derived propagators, not incurring any performance penalties compared to dedicated, handwritten propagators.

Experimental setup. The experiments are based on Gecode 3.1.0, compiled using the GNU C++ compiler gcc 4.3.2, on an Intel Pentium IV at 2.8 GHz running Linux. Runtimes are the average of 25 runs, with a coefficient of deviation less than 2.5% for all benchmarks. All example programs are available in the Gecode distribution. Table 2 shows the figures for the unmodified Gecode 3.1.0 (pure integer models above, models with integer and set variables below the horizontal line), and results will be given relative to these numbers. For example, a runtime of 130% means that the example needs 30% more time, while 50% means that it is twice as fast as in Gecode 3.1.0. The column *time* shows the runtime, *mem.* the peak allocated memory, *failures* the number of failures during search, and *propagations* the number of propagator invocations.

Table 3: Relative performance of decomposition, compared to views

Benchmark	*time %*	*mem. %*	*propagations %*
Alpha (naive)	412.88	360.87	673.83
BIBD (7-3-60)	308.80	211.94	256.12
Eq-20	590.35	700.00	704.57
Partition (32)	135.61	113.58	136.40
Perfect Square	114.46	109.67	104.42
Queens (Dom, 10)	173.32	100.00	519.68
Queens (Dom, 100)	140.60	103.11	2 371.86
Social Golfers (8-4-9)	335.89	234.82	160.22
Social Golfers (5-3-7)	217.28	190.69	150.58
Hamming Codes (20-3-32)	113.81	104.66	99.65
Steiner Triples (9)	132.79	100.00	101.76
Sudoku (Set, 1)	166.18	100.00	110.38
Sudoku (Set, 4)	152.82	110.81	107.06
Sudoku (Set, 5)	143.63	100.00	105.47

As many of the experimental results rely on the optimization capabilities of the used C++ compiler, we verified that all experiments yield similar results with the Microsoft Visual Studio 2008 C++ compiler.

7.1 Views Versus Decomposition

In order to evaluate whether deriving propagators is worth the effort in the first place, this set of experiments compares derived propagators with their decompositions, revealing a significant overhead of the latter.

Table 3 shows the results of these experiments. For *Alpha* and *Eq-20*, linear equations with coefficients are decomposed. For *Queens 100*, we replace the special *all-different*-with-offsets by its decomposition into an *all-different* propagator and binary equality-with-offset propagators. In *BIBD* and *Perfect Square*, we decompose ternary Boolean propagators, implementing $x \wedge y \leftrightarrow z$ as $\neg x \vee \neg y \leftrightarrow \neg z$ in *BIBD*, and $x \vee y \leftrightarrow z$ as $\neg x \wedge \neg y \leftrightarrow \neg z$ in *Perfect Square*. In the remaining examples, we decompose a set intersection into complement and union propagators.

Some integer examples show a significant overhead of around six times the runtime and memory when decomposed. The overhead of most set examples as well as *Perfect Square* is moderate, partly because no additional variable was introduced if the model already contained its complement or negation. As to be expected, decomposition often needs significantly more propagation steps, but as the additional steps are performed by cheap propagators (like $x = y + i$ or $x = \neg y$), the runtime effect is less drastic. *Queens 100* is an extreme case, where 23 times the propagation steps only cause 40% more runtime. The reason is that the scheduling order does not take advantage of the fact that the decompositions are Berge-acyclic as discussed in Section 3.5. *Partition 32* has a single linear equation with coefficients, several linear equations with unit coefficients, multiplications, and a single *all-different*. Replacing the linear equation by its decomposition has little effect on the runtime (35% overhead).

7.2 Impact of Derivation Techniques

The techniques presented in Section 4 have different impacts on the performance of the derived propagators.

Table 4: Relative performance of minus views

Benchmark	*time %*	*prop. %*	*Benchmark*	*time %*	*prop. %*
All-Interval (50)	100.00	100.00	Partition (32)	131.83	135.95
All-Interval (100)	100.52	100.00	Queens (10)	98.32	100.00
Alpha (naive)	101.81	100.00	Queens (Dom, 10)	107.63	100.00
Golomb Rulers (Bnd, 10)	99.36	100.01	Queens (100)	97.83	100.00
Golomb Rulers (Dom, 10)	107.77	100.18	Queens (Dom, 100)	95.62	100.00
Graph Coloring	107.19	99.84	Sorting (400)	105.13	100.00

Generalization and specialization. These techniques can be implemented without any performance overhead compared to a handwritten propagator. This is not surprising as the only potential overhead could be that a function call is not resolved at compile time. For example, a thorough inspection of the code generated by the GNU C++ compiler and the Microsoft Visual Studio C++ compiler shows that they are able to fully inline the operations of offset and scale views.

Transformation and type conversion. These techniques can incur an overhead compared to a dedicated implementation, as the transformations performed by the views can sometimes not be removed by compiler optimizations, and type conversions may be costly if the data structures for the variable domains differ significantly.

For example, a propagator instantiated with two minus views of variables x and y may include a comparison, $(-x) < (-y)$. Due to the invariants guaranteed by views, this is equivalent to $y < x$, saving two negations. However, the asymmetry in the two's complement representation of integers prevents the compiler from performing this optimization. As an experiment to evaluate this effect, we instantiated an *all-different* propagator with minus views. The resulting derived propagator of course implements the same constraint, but incurs the overhead of negation. Similarly, we replaced the max propagator in the *Sort* example with a min (where the propagator for min is derived from the propagator for max) and negated all parameters. According to the results in Table 4, the overhead is often negligible, and only exceeds 5% in examples that use the domain-complete *all-different* propagator (*Graph Coloring*, *Golomb Rulers Dom* and *Queens Dom*) or predominantly min propagators (*Sort*). *Queens Dom 100* does not show the effect as the runtime is dominated by search. Using minus views can result in different propagator scheduling. The *Partition* example shows this behavior, where the increase in propagation steps results in increased runtime.

It is interesting to note that the domain-complete *all-different* propagator, when instantiated with minus views, requires a cache iterator for sequence reversal (as discussed in Section 5.4). Surprisingly, the overhead of minus views is largely independent of the use of cache iterators which is confirmed in Section 7.4.

Other transformations are translated optimally, such as turning $(-x)-(-y)$ into $y-x$. Boolean negation views also lead to optimal code, as they do not compute $1-x$ for a Boolean variable x, but instead swap the positive and negative operations.

Set-valued transformations can result in non-optimal code. For example, a propagator for ternary intersection, $x = y \cap z$, will include an inference $x.\texttt{adjglb}(y.\texttt{glb}() \cap z.\texttt{glb}())$. To derive a propagator for $x = y \cup z$, we instantiate the intersection propagator with complement views for x, y, and z, yielding the following inference:

$$\overline{x}.\texttt{adjglb}(\overline{y}.\texttt{glb}() \cap \overline{z}.\texttt{glb}())$$

Table 5: Relative performance of views compared to dedicated set propagators

Benchmark	*time %*	*Benchmark*	*time %*
Social Golfers (8-4-9)	166.31	Sudoku (Set, 1)	167.44
Social Golfers (5-3-7)	148.83	Sudoku (Set, 4)	151.74
Hamming Codes (20-3-32)	129.11	Sudoku (Set, 5)	142.83
Steiner Triples (9)	127.85		

Table 6: Relative performance of virtual method calls

Benchmark	*time %*	*Benchmark*	*time %*
All-Interval (50)	182.63	Social Golfers (8-4-9)	148.37
All-Interval (100)	113.20	Social Golfers (5-3-7)	138.95
Alpha (naive)	153.59	Hamming Codes (20-3-32)	131.37
BIBD (7-3-60)	138.50	Steiner Triples (9)	149.08
Eq-20	211.69	Sudoku (Set, 1)	119.56
Golomb Rulers (Bnd, 10)	220.01	Sudoku (Set, 4)	118.78
Golomb Rulers (Dom, 10)	170.13	Sudoku (Set, 5)	119.17
Graph Coloring	104.29		
Magic Sequence (Smart, 500)	136.58		
Magic Sequence (GCC, 500)	226.64		
Partition (32)	187.89		
Perfect Square	130.64		
Queens (10)	133.81		
Queens (100)	160.79		

which amounts to computing

$$x.\mathtt{adjlub}(\overline{\overline{y.\mathtt{lub}()} \cap \overline{z.\mathtt{lub}()}})$$

It would be more efficient to implement the equivalent $x.\mathtt{adjlub}(y.\mathtt{lub}() \cup z.\mathtt{lub}())$ because this requires three set operations less. Unfortunately, no compiler will find this equivalence automatically, as it requires knowledge about the semantics of the set operations. Table 5 compares a dedicated propagator for the constraint $x \cap y = z$ with a version using complement views and a propagator for $x \cup y = z$. The overhead of 27% to 67% does not render views useless for set variables, but it is nevertheless significant.

7.3 Templates Versus Virtual Methods

As suggested in Section 5, in C++, compile-time polymorphism using templates is far more efficient than virtual method calls. To evaluate this, we changed the basic operations of integer variables to be virtual methods, such that view operations need one virtual method call. In addition, all operations that use templates (and can therefore not be made virtual in C++) have been changed so that they cannot be inlined, to simulate virtual method calls. This is a conservative approximation of the actual cost of fully virtual views. The results of these experiments appear in Table 6. Virtual method calls cause a runtime overhead between 4% and 127% for the integer examples (left table), and 18% to 49% for the set examples (right table). The runtime overhead for set examples is lower as the basic operations on set variables are considerably more expensive than the basic operations on integer variables.

Table 7: Relative performance of cache iterators

Benchmark	*time %*	*Benchmark*	*time %*
All-Interval (50)	102.48	Social Golfers (8-4-9)	522.62
All-Interval (100)	101.17	Social Golfers (5-3-7)	450.15
Golomb Rulers (Bnd, 10)	100.51	Hamming Codes (20-3-32)	297.38
Golomb Rulers (Dom, 10)	128.98	Steiner Triples (9)	304.97
Graph Coloring	144.58	Sudoku (Set, 1)	459.85
Magic Sequence (GCC, 500)	103.56	Sudoku (Set, 4)	483.27
Queens (Dom, 10)	187.36	Sudoku (Set, 5)	436.92
Queens (Dom, 100)	155.62		

7.4 Iterators Versus Temporary Data Structures

The following experiments show that using range iterators improves the efficiency of propagators, compared to the use of explicit set data structures for temporary results.

For the experiments, temporary data structures have been emulated by wrapping all iterators in a cache iterator as described in Section 5.4. Table 7 shows the results. For integer propagators that perform the safe iterator-based domain operations introduced in Section 5.4, computing with temporary data structures results in 28% to 87% overhead (*Golomb Rulers Dom, Graph Coloring, Queens Dom*). For set propagators, which make much more use of iterators than integer propagators, the overhead becomes prohibitive, resulting in up to 4.8 times the runtime. The memory consumption does not increase, because iterators are not stored, and only few iterators are active at a time.

8 Conclusion

The paper has developed views for deriving propagator variants. Such variants are ubiquitous, and the paper has shown how to systematically derive propagators using different types of views, corresponding to techniques such as transformation, generalization, specialization, and type conversion.

Based on a formal, implementation independent model of propagators and views, the paper has identified fundamental properties of views that result in *perfect* derived propagators. The paper has shown that a derived propagator inherits correctness and domain completeness from its original propagator, and bounds completeness given additional properties of the used views.

The paper has presented an implementation architecture for views based on *parametricity*. The propagator implementation is kept parametric over the type of view that is used, so that deriving a propagator amounts to instantiating a parametric propagator with the proper views. This implementation architecture is an orthogonal layer of abstraction that can be implemented on top of any constraint solver.

An empirical evaluation has shown that views have proven invaluable for the implementation of Gecode, saving huge amounts of code to be written and maintained. Furthermore, deriving propagators using templates in C++ has been shown to yield competitive (in most cases optimal) performance compared to dedicated handwritten propagators. The experiments have also clarified that deriving propagators is vastly superior to decomposing the constraints into additional variables and simple propagators.

Acknowledgments. We thank Mikael Lagerkvist, Gert Smolka, and Thibaut Feydy for fruitful discussions. Christian Schulte has been partially funded by the Swedish

Research Council (VR) under grant 621-2004-4953.

References

[1] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, Cambridge, UK, 2003.

[2] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based Scheduling*. International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2001.

[3] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[4] Nicolas Beldiceanu, Irit Katriel, and Sven Thiel. Filtering algorithms for the same constraint. In Régin and Rueher [29], pages 65–79.

[5] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206, Southampton, UK, September 1997. Springer.

[6] CHOCO. `choco-solver.net`, 2009.

[7] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In Abdul Sattar and Byeong-Ho Kang, editors, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *LNCS*, pages 49–58. Springer, 2006.

[8] ECL^{i}PS^e. `www.eclipse-clp.org`, 2009.

[9] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518, Santa Margherita Ligure, Italy, May 2004. Springer.

[10] Gecode. `www.gecode.org`, 2009.

[11] Ian P. Gent, editor. *Fifteenth International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, Lisbon, Portugal, September 2009. Springer. To appear.

[12] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In Frédéric Benhamou, editor, *Twelfth Internation Conference on Principles and Practice of Constraint Programming*, volume 4204 of *LNCS*, pages 182–197, Nantes, France, September 2006. Springer.

[13] Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *International Symposium on Logic Programming*, pages 339–358, Ithaca, NY, USA, 1994. MIT Press.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[15] Warwick Harvey and Peter J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 7:173–207, 2003.

[16] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[17] ILOG Solver, part of ILOG CP. `www.ilog.com/products/cp`, 2009.

[18] François Laburthe. Choco: Implementing a CP kernel. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 71–85, September 2000.

[19] Mikael Z. Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints BPPC'08*, 2008.

[20] Mikael Z. Lagerkvist and Christian Schulte. Propagator groups. In Gent [11]. To appear.

[21] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[22] Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[23] The Mozart programming system. `www.mozart-oz.org`, 2009.

[24] Simon L. Peyton Jones. Haskell 98. *Journal of Functional Programming*, 13(1), 2003.

[25] Jean-Françcois Puget. PECOS: A high level constraint programming language. In *Proceedings of the first Singapore international conference on Intelligent Systems (SPICIS)*, pages 137–142, Singapore, 1992.

[26] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press.

[27] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, November 1994.

[28] Claude-Guy Quimper. *Efficient Propagators for Global Constraints*. PhD thesis, University of Waterloo, Canada, 2006.

[29] Jean-Charles Régin and Michel Rueher, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *LNCS*, Nice, France, April 2004. Springer.

[30] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[31] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[32] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space? *Transactions on Programming Languages and Systems*, 27(3):388–425, May 2005.

[33] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.

[34] Christian Schulte and Guido Tack. Weakly monotonic propagators. In Gent [11]. To appear.

[35] SICStus Prolog, 2009. `www.sics.se/sicstus/`.

[36] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):293–316, 1998.

[37] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In Régin and Rueher [29], pages 335–347.