Perfect Derived Propagators

Christian Schulte[1] and Guido Tack[2]

[1] ICT, KTH - Royal Institute of Technology, Sweden, cschulte@kth.se
[2] PS Lab, Saarland University, Saarbrücken, 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 equations both with and without coefficients? Constraint variants are ubiquitous: implementing them requires considerable (if not prohibitive) effort and decreases maintainability, but will deliver better performance.

This paper shows how to use variable views, previously introduced for an implementation architecture, 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 channeling are developed for several variable domains. We evaluate the massive impact of derived propagators. Without derived propagators, Gecode would require 140 000 rather than 40 000 lines of code for propagators.

1 Introduction

When implementing a propagator for a constraint, one typically needs to decide whether to also implement some of its variants. For example, when implementing a propagator for $\max_{i=1}^{n} x_i = y$, should one also implement $\min_{i=1}^{n} x_i = y$? When implementing the linear equation $\sum_{i=1}^{n} a_i x_i = c$ for integer variables x_i and integers a_i and c, should one also implement $\sum_{i=1}^{n} x_i = c$ for better performance? When implementing the reified linear equation $(\sum_{i=1}^{n} x_i = c) \Leftrightarrow b$, should one also implement its almost identical algebraic variant $(\sum_{i=1}^{n} x_i \neq c) \Leftrightarrow b$?

Implementing inflates code and documentation. Not implementing increases space and runtime: by using more general propagators or by decomposing into several other constraints. Worse, given the potential code explosion, one may be able to only implement some variants (say, minimum and maximum). Other variants important for performance (say, minimum and maximum for two variables) may be infeasible due to excessive programming and maintenance effort.

Here, we follow a third approach: we derive propagators from already existing propagators using variable views. In [12], we introduced an implementation architecture for variable views to reuse generic propagators without performance penalty. This architecture has been implemented in Gecode [5], and is in fact essential for the system, as it saves approximately 100 000 lines of code. Due to the massive use of views in Gecode, it is vital to develop a model that allows us to prove that derived propagators have the desired properties.

In this paper, we argue that propagators that are derived using variable views are indeed *perfect*: they are not only perfect for performance, we prove that they inherit all essential properties such as correctness and completeness from their original propagator.

Last but not least, we show common techniques for deriving propagators with views and demonstrate their wide applicability. In Gecode, every propagator implementation is reused 3.6 times on average. Without views, Gecode would feature 140 000 rather than 40 000 lines of propagator implementation to be written, tested, and maintained.

Variable views. Consider a bounds consistent propagator for $\max(x, y) = z$. Assume that $\overline{x}$ ($\underline{x}$) returns the maximum (minimum) of the finite domain variable x, whereas $\overline{x} \leftarrow n$ ($\underline{x} \leftarrow n$) adjusts the maximum (minimum) value of x to $\min(\overline{x}, n)$ ($\max(\underline{x}, n)$), only taking variable bounds into account. The propagator is implemented by performing the following operations on its variables:

$$\overline{x} \leftarrow \overline{z} \qquad \overline{y} \leftarrow \overline{z} \qquad \overline{z} \leftarrow \max(\overline{x}, \overline{y}) \qquad \underline{z} \leftarrow \max(\underline{x}, \underline{y})$$

Given three more propagators for $x' = -x$, $y' = -y$, and $z' = -z$, we could propagate the constraint $\min(x', y') = z'$. In contrast to this *decomposition*, we propose to use generic propagators that perform operations on views rather than variables. Views provide the same interface (set of operations) as variables while enabling additional transformations. For example, an operation on a minus view x' on a variable x behaves as if executed on $-x$: $\overline{x'}$ is defined as $-\underline{x}$ and $\overline{x'} \leftarrow n$ is defined as $\underline{x} \leftarrow -n$. With views, the implementation of the maximum propagator can be reused: we *derive* a propagator for the minimum constraint by instantiating the maximum propagator with minus views for its variables.

The feasibility of variable views rests on today's programming languages' support for generic (or polymorphic) constructions (for example, templates in C++) and that the simple transformations provided by views are optimized away.

Contributions. This paper contributes an implementation independent model for views and derived propagators, techniques for deriving propagators, and an evaluation that shows that views are widely applicable, drastically reduce programming effort, and are more efficient than decomposition.

More specifically, the key contribution is the identification of properties of views that are essential for deriving *perfect* propagators. To this end, 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 variable domains to variable domains). This model yields all the desired results: derived propagators are indeed propagators; derived propagators 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 properties.

After establishing the fundamental results, we address further properties of derived propagators such as idempotence, subsumption, and events. Finally, we clarify the connection between derived propagators and path consistency when regarding views as binary constraints.

We introduce techniques for deriving propagators that use views for specialization and generalization of propagators, channeling between variable domains, and general domain-specific transformations. We show how to apply these techniques for different variable domains using various views. We provide a breakdown of how successful the use of derived propagators has been for Gecode.

Overview. The next section introduces the basic notions we will use. Sect. 3 presents views and derived propagators and proves fundamental properties like correctness and completeness. The following three sections develop techniques for deriving propagators: transformation, generalization, specialization, and channeling. Sect. 7 presents extensions of the model, and Sect. 8 discusses its limitations. Sect. 9 provides empirical evidence that views are useful in practice.

2 Preliminaries

This section sets the stage for the paper with definitions of the basic concepts.

Variables and constraints. We assume a finite set of variables $\mathit{Var} = \{x_1, \ldots, x_n\}$ and a finite set of values Val. Constraints are characterized by assignments $a \in \mathit{Asn}$ that map variables to values: $\mathit{Asn} = \mathit{Var} \to \mathit{Val}$. A constraint $c \in \mathit{Con}$ is a relation over the variables, represented as the set of all assignments that satisfy the constraint, $\mathit{Con} = 2^{\mathit{Asn}}$. We base constraints on full assignments, defined for all variables in Var. However, for typical constraints, only a subset $\mathrm{vars}(c)$ of the variables is *significant*; the constraint is the full relation for all $x \notin \mathrm{vars}(c)$. We write a constraint in extension ($c = \{(x \mapsto 0, y \mapsto 1), (x \mapsto 1, y \mapsto 2)\}$) or intensionally ($c \equiv x < y$).

Domains. Constraints are implemented by propagators over domains, which are constructed as follows. A *domain* $d \in \mathit{Dom}$ maps each variable to a finite set of possible values, the *variable domain* $d(x) \subseteq \mathit{Val}$.

A domain d can be identified with a set of assignments $d \in 2^{\mathit{Asn}}$. We can therefore treat domains as constraints. In particular, for any assignment a, $\{a\}$ is a domain as well as a constraint. We simply write *domain* for domains and variable domains when there is no risk of confusion.

A domain d_1 is *stronger* than a domain d_2 (written $d_1 \subseteq d_2$), iff for all variables x, $d_1(x) \subseteq d_2(x)$. By $\mathrm{dom}(c)$ we refer to the strongest domain including all valid assignments of a constraint, defined as $\min\{d \in \mathit{Dom} \mid c \subseteq d\} = \{a \mid \forall x\ \exists b \in c.\ a(x) = b(x)\}$. The minimum exists as domains are closed under intersection, and the definition is non-trivial because not every constraint can be captured by a domain. Now, for a constraint c and a domain d, $\mathrm{dom}(c \cap d)$ refers to removing all values from d not supported by the constraint c.

Propagators. Propagators serve here as implementations of constraints. They are sometimes also referred to as constraint narrowing operators or filter functions. A propagator is a function $p \in \mathit{Dom} \to \mathit{Dom}$ that is contracting ($p(d) \subseteq d$) and monotone ($d' \subseteq d \Rightarrow p(d') \subseteq p(d)$). Idempotence is not required.

Propagators are contracting, they only remove values from variable domains. For an assignment a, a propagator p hence has only two options: accept it ($p(\{a\}) = \{a\}$), or reject it ($p(\{a\}) = \emptyset$). Monotonicity guarantees that if some domain d contains an assignment $a \in d$ that p accepts, then p will not remove a from d: $a \in p(d)$. The propagator therefore behaves like a characteristic function for the set of accepted assignments. This set is the *associated constraint of p*.

We say that a propagator p *implements its associated constraint* $c_p = \{a \in \mathit{Asn} \mid p(\{a\}) = \{a\}\}$. Monotonicity implies that for any domain d, we have $\mathrm{dom}(c_p \cap d) \subseteq p(d)$: no solution of c_p is ever removed by p. We say that p is *sound* for any $c \subseteq c_p$ and *weakly complete* for any $c' \supseteq c_p$ (meaning that it accepts all assignments in c and rejects all assignments not in c'). For any constraint c, we can find at least one propagator p such that $c = c_p$. Typically, there are several propagators, differing by *propagation strength* (see Sect. 3).

Our definitions of soundness and different notions of completeness for propagators are based on and equivalent to Benhamou's [2] and Maher's [9]. We specify *what* is computed by constraint propagation and not *how*. Approaches for performing constraint propagation can be found in [2, 1, 11].

3 Views and Derived Propagators

We now introduce our central concepts, views and derived propagators.

A *view* on a variable x is an injective function $\varphi_x \in \mathit{Val} \to \mathit{Val}'$, mapping values from Val to values from a possibly different set Val'. We lift a family of views φ_x (one for each $x \in \mathit{Var}$) point-wise to assignments as follows: $\varphi_{Asn}(a)(x) = \varphi_x(a(x))$. Finally, given a family of views lifted to assignments, we define a view $\varphi \in \mathit{Con} \to \mathit{Con}$ on constraints as $\varphi(c) = \{\varphi_{Asn}(a) \mid a \in c\}$. The inverse of that view is defined as $\varphi^-(c) = \{a \in \mathit{Asn} \mid \varphi_{Asn}(a) \in c\}$.

In the implementation, a view on x presents the same interface as x, but applies transformations when a propagator adjusts or accesses the domain of x through the view. In our model, φ performs the transformations for accessing, and φ^- for adjusting the variable domains. Views can now be composed with a propagator: a *derived propagator* is defined as $\widehat{\varphi}(p)(d) = \varphi^-(p(\varphi(d)))$, or, using function composition, as $\widehat{\varphi}(p) = \varphi^- \circ p \circ \varphi$.

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

It is usually easier to think about the other direction: $\varphi(c') \subseteq c$. 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 $\varphi_y(v) = 2v$. We have a subset relation because some tuples of c may be ruled out by φ. For instance, with φ defined as above, there is no assignment a such that $\varphi_{Asn}(a)(y) = 3$, but the assignment $(x \mapsto 3, y \mapsto 3)$ is in c.

This example also makes clear why the set Val' is allowed to differ from Val. In this particular case, Val' has to contain all multiples of 2 of elements in Val.

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 Val.

Given the assignment $a = (x \mapsto 2, y \mapsto 1)$, we first apply φ_{Asn} and get $\varphi_{Asn}(a) = (x \mapsto 2, y \mapsto 2)$. 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 $\varphi_{Asn}(a') = (x \mapsto 1, y \mapsto 4)$, rejected ($p(\{\varphi_{Asn}(a')\}) = \emptyset$), and the empty domain is mapped to the empty domain by φ^-. The propagator $\widehat{\varphi}(p)$ implements $\varphi^-(c)$. □

Views and derived propagators satisfy a number of essential properties:

1. A derived propagator $\widehat{\varphi}(p)$ is in fact a propagator.
2. The associated constraint of $\widehat{\varphi}(p)$ is $\varphi^-(c_p)$.
3. A view φ preserves contraction of a propagator p: If $p(\varphi(d)) \subset \varphi(d)$, then $\widehat{\varphi}(p)(d) \subset d$. This property makes sure that if the propagator makes an inference, then this inference will actually be reflected in a domain change.

In the following, we will prove these properties. For the proofs, we employ some direct consequences of the definitions of views and derived propagators: (1) φ and φ^- are monotone by construction; (2) $\varphi^- \circ \varphi = \mathrm{id}$ (the identity function); (3) $|\varphi(\{a\})| = 1$, $\varphi(\emptyset) = \emptyset$; (4) for any view φ and domain d, we have $\varphi(d) \in Dom$ and $\varphi^-(d) \in Dom$ (as views are defined point-wise).

Theorem 1. A derived propagator is a propagator: for all propagators p and views φ, $\widehat{\varphi}(p)$ is a monotone and contracting function in $Dom \to Dom$. □

Proof. The derived propagator is well-defined because both $\varphi(d)$ and $\varphi^-(d)$ are domains (see (4) above). Monotonicity is obvious, as compositions of monotone functions are monotone. For contraction, we have $p(\varphi(d)) \subseteq \varphi(d)$ as p is contracting. By monotonicity of φ^-, we know that $\varphi^-(p(\varphi(d))) \subseteq \varphi^-(\varphi(d))$. As $\varphi^- \circ \varphi = \mathrm{id}$, we have $\varphi^-(p(\varphi(d))) \subseteq d$, which proves that $\widehat{\varphi}(p)$ is contracting. In summary, for any propagator p, $\widehat{\varphi}(p) = \varphi^- \circ p \circ \varphi$ is a propagator. ■

Theorem 2. If p implements c_p, then $\widehat{\varphi}(p)$ implements $\varphi^-(c_p)$. □

Proof. As p implements c_p, we know $p(\{a\}) = c_p \cap \{a\}$ for all assignments a. With $|\varphi(\{a\})| = 1$, 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 $\varphi^-(p(\varphi(\{a\}))) = \emptyset = \{a\} \cap \varphi^-(c_p)$. Case $\varphi(\{a\})$: As $\varphi^- \circ \varphi = \mathrm{id}$, we have $\varphi^-(p(\varphi(\{a\}))) = \{a\}$. Furthermore:

$$\begin{array}{rcl} c_p \cap \varphi(\{a\}) = \varphi(\{a\}) & \Rightarrow & \exists b \in c_p.\ b = \varphi(a) \\ \Rightarrow a \in \{a' \in Asn \mid \varphi(a') \in c_p\} & \Rightarrow & a \in \varphi^-(c_p) \end{array}$$

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

Theorem 3. Views preserve contraction: for any domain d, if $p(\varphi(d)) \subseteq \varphi(d)$, then $\widehat{\varphi}(p)(d) \subset d$. □

Proof. Recall the definition of $\varphi^{-}(c)$ as $\{a \in Asn \mid \varphi_{Asn}(a) \in c\}$. It clearly follows that $|\varphi^{-}(c)| \leq |c|$. Similarly, we know that $|\varphi(c)| = |c|$. From $p(\varphi(d)) \subset \varphi(d)$, we know that $|p(\varphi(d))| < |\varphi(d)|$. Together, this yields $|\widehat{\varphi}(p)(d)| < |\varphi(d)| = |d|$. We have already seen in Theorem 1 that $\widehat{\varphi}(p)(d) \subseteq d$, so we can conclude that $\widehat{\varphi}(p)(d) \subset d$. ■

Completeness. Weak completeness, as introduced above, is the minimum required for a constraint solver to be complete. A weakly complete propagator does not have to prune variable domains, it only has to check if an assigned domain is a solution of the constraint. The success of constraint propagation however crucially depends on strong propagators that prune variable domains.

The strongest possible inference that a single propagator can do establishes *domain consistency* (also known as *generalized arc consistency*): a domain d is domain consistent for a constraint c, iff for all variables x_i and all values $v_i \in d(x_i)$, there exist values $v_j \in d(x_j)$ for all other variables x_j such that the assignment $(x_1 \mapsto v_1, \ldots, x_i \mapsto v_i, \ldots, x_n \mapsto v_n)$ is a solution of c.

A propagator is *domain complete* (or simply complete) for a constraint c if it establishes domain consistency. More formally, a propagator p is complete for a constraint c iff for all domains d, we have $p(d) \subseteq \text{dom}(c \cap d)$. A complete propagator thus removes all assignments from d that are inconsistent with c.

We will now prove that propagators derived from complete propagators are also complete. In Sect. 5, we will extend this result to weaker notions of completeness, such as bounds($\mathbb{Z}$) and bounds($\mathbb{R}$) completeness.

For this proof, we need two auxiliary definitions. A constraint c is a φ *constraint* iff for all $a \in c$, there is a $b \in Asn$ such that $a = \varphi_{Asn}(b)$. A view φ is *dom injective* iff $\varphi^{-}(\text{dom}(c)) = \text{dom}(\varphi^{-}(c))$ for all φ constraints c.

For the completeness proof, we need a lemma that states that any view is dom injective.

Proof. By definition of φ^{-} and $\text{dom}(\cdot)$, we have $\varphi^{-}(\text{dom}(c)) = \{a \in Asn \mid \forall x. \exists b \in c. \varphi_{Asn}(a)(x) = b(x)\}$. As c is a φ constraint, 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 $\varphi_{Asn}(b') = b$. Therefore, we get $\{a \in Asn \mid \forall x. \exists b' \in \varphi^{-}(c). a(x) = b'(x)\} = \text{dom}(\varphi^{-}(c))$. ■

Furthermore, we need a lemma that states that views commute with set intersection: 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 $\varphi^{-}(c_1 \cap c_2) = \{a \in Asn \mid \varphi_{Asn}(a) \in c_1 \wedge \varphi_{Asn}(a) \in c_2\}$. As φ_{Asn} is a function, this is equal to $\{a \in Asn \mid \varphi_{Asn}(a) \in c_1\} \cap \{a \in Asn \mid \varphi_{Asn}(a) \in c_2\} = \varphi^{-}(c_1) \cap \varphi^{-}(c_2)$. ■

Theorem 4. If p is complete for c, then $\widehat{\varphi}(p)$ is complete for $\varphi^{-}(c)$. □

Proof. By monotonicity of φ and completeness of p, we know that $\varphi^- \circ p \circ \varphi(d) \subseteq \varphi^-(\mathrm{dom}(c \cap \varphi(d)))$. We now use the fact that φ^- is dom injective and commutes with set intersection:

$$\varphi^-(\mathrm{dom}(c \cap \varphi(d))) = \mathrm{dom}(\varphi^-(c \cap \varphi(d))) =$$
$$\mathrm{dom}(\varphi^-(c) \cap \varphi^-(\varphi(d))) = \mathrm{dom}(\varphi^-(c) \cap d)$$

∎

4 Boolean Variables: Transformation

This section discusses views and derived propagators for Boolean variables where $\mathit{Val} = \{0, 1\}$. Not surprisingly, the only view apart from identity for Boolean variables captures negation. That is, using a *negation view* on x defines $\varphi_x(v) = 1 - v$ for $x \in \mathit{Var}$ and $v \in \mathit{Val}$.

Negation views are more widely applicable than one would initially believe. They demonstrate how views can be used systematically to obtain implementations of constraint variants by *transformation*.

Boolean connectives. The immediate 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 in models, it pays off to optimize frequently occurring cases. One important propagator is disjunction $\bigvee_{i=1}^{n} x_i = y$ for arbitrarily many variables; again conjunction can be derived with negation views on the x_i and on y. Another important propagator is for the constraint $\bigvee_{i=1}^{n} x_i = 1$, stating that the disjunction must be true. A propagator for this constraint is essential as the constraint occurs frequently and as it can be implemented efficiently using watched literals, see for example [6]. With views and derived propagators 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 with the following transformation:

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

Reification. Many reified constraints (such as $(\sum_{i=1}^{n} x_i = c) \Leftrightarrow b)$ also exist in a negated version (such as $(\sum_{i=1}^{n} x_i \neq c) \Leftrightarrow b)$. 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 entailed must be factorized out and combined differently for the two variants.

5 Integer Variables: Generalization, Bounds Consistency, Specialization

Common views for finite domain integer variables capture linear transformations of the integer values. In [12], the following views are introduced for a variable x and values v: a *minus view* on x is defined as $\varphi_x(v) = -v$, 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 $\varphi_x(v) = a \cdot v$.

Propagators for integer variables offer a greater degree of freedom concerning their level of completeness. While Boolean propagators most often will be domain complete, bounds completeness is important for integer propagators. Before we discuss transformation and generalization techniques for deriving integer propagators, we study how bounds completeness is affected by views.

Bounds consistency and bounds completeness. There are several different notions of bounds consistency in the literature (see [4] for an overview). For our purposes, we distinguish bounds($\mathcal{D}$), bounds($\mathbb{Z}$), and bounds($\mathbb{R}$) consistency:

- A domain d is bounds($\mathcal{D}$) *consistent* for a constraint c, iff for all variables x_i there exist $v_j \in d(x_j)$ for all other variables x_j such that $\{x_1 \mapsto v_1, \ldots, x_i \mapsto \min(d(x_i)), \ldots, x_n \mapsto v_n\} \in c$ and analogously for $x_i \mapsto \max(d(x_i))$.
- A domain d is bounds($\mathbb{Z}$) *consistent* for a constraint c, iff for all variables x_i, there exist integers v_j with $\min(d(x_j)) \leq v_j \leq \max(d(x_j))$ for all other variables x_j such that $\{x_1 \mapsto v_1, \ldots, x_i \mapsto \min(d(x_i)), \ldots, x_n \mapsto v_n\} \in c$ and analogously for $x_i \mapsto \max(d(x_i))$.
- A domain d is bounds($\mathbb{R}$) *consistent* for a constraint c, iff for all variables x_i, there exist real numbers $v_j \in \mathbb{R}$ with $\min(d(x_j)) \leq v_j \leq \max(d(x_j))$ for all other variables x_j such that $\{x_1 \mapsto v_1, \ldots, x_i \mapsto \min(d(x_i)), \ldots, x_n \mapsto v_n\} \in c_{\mathbb{R}}$ and analogously for $x_i \mapsto \max(d(x_i))$, where $c_{\mathbb{R}}$ is c relaxed to $\mathbb{R}$ (for constraints like arithmetics where relaxation makes sense).

A propagator p is bounds(X) complete for its associated constraint c_p, iff $p(d)$ is bounds(X) consistent for c_p for every domain d that is a fixpoint of p. We use an equivalent definition based on the *strongest convex domain* that contains a constraint, $\text{conv}(c) = \min\{d \in \mathit{Dom} \mid c \subseteq d \text{ and } d \text{ convex}\}$. A convex domain maps each variable to an interval, so that $\text{conv}(c)(x) = \{\min_{a \in c}(a(x)), \ldots, \max_{a \in c}(a(x))\}$. Note that $\text{conv}(c)$ is weaker than the strongest domain that contains c: $\text{conv}(c) \supseteq \text{dom}(c)$ for all constraints c. In the same way as Benhamou [2] and Maher [9], we define

- p is bounds($\mathcal{D}$) complete for c iff $p(d) \subseteq \mathrm{conv}(c \cap d)$.
- p is bounds($\mathbb{Z}$) complete for c iff $p(d) \subseteq \mathrm{conv}(c \cap \mathrm{conv}(d))$.
- p is bounds($\mathbb{R}$) complete for c iff $p(d) \subseteq \mathrm{conv}(c_{\mathbb{R}} \cap \mathrm{conv}_{\mathbb{R}}(d))$, where $\mathrm{conv}_{\mathbb{R}}(d)$ is the convex hull of d in $\mathbb{R}$, and $c_{\mathbb{R}}$ is c relaxed to $\mathbb{R}$.

Bounds completeness of derived propagators. Theorem 4 states that propagators derived from domain complete propagators are domain complete. A similar theorem holds for bounds completeness, if views commute with $\mathrm{conv}(\cdot)$ in the following ways:

A view φ is *interval injective* iff $\varphi^{-}(\mathrm{conv}(c)) = \mathrm{conv}(\varphi^{-}(c))$ for all φ constraints c. It is *interval bijective* iff it is interval injective and $\varphi(\mathrm{conv}(d)) = \mathrm{conv}(\varphi(d))$ for all domains d.

Proving bounds completeness of derived propagators is now similar to proving domain completeness. We only formulate bounds($\mathbb{Z}$) completeness.

Theorem 5. If p is bounds($\mathbb{Z}$) complete for c and φ is interval bijective, then $\widehat{\varphi}(p)$ is bounds($\mathbb{Z}$) complete for $\varphi(c)$. □

Proof. By monotonicity of φ and bounds($\mathbb{Z}$) completeness of p, we know that $\varphi^{-} \circ p \circ \varphi(d) \subseteq \varphi^{-}(\mathrm{conv}(c \cap \mathrm{conv}(\varphi(d))))$. We now use the fact that both φ and φ^{-} commute with conv and intersection:

$$\varphi(\mathrm{conv}(c \cap \mathrm{conv}(\varphi^{-1}(d)))) = \varphi(\mathrm{conv}(c \cap \varphi^{-1}(\mathrm{conv}(d)))) =$$
$$\mathrm{conv}(\varphi(c \cap \varphi^{-1}(\mathrm{conv}(d)))) = \mathrm{conv}(\varphi(c) \cap \varphi(\varphi^{-1}(\mathrm{conv}(d)))) =$$
$$\mathrm{conv}(\varphi(c) \cap \mathrm{conv}(d))$$
■

The proof for bounds($\mathcal{D}$) is analogous, but we only require interval injectivity for the view. With an interval injective view, one can also derive bounds($\mathbb{R}$) complete propagators from bounds($\mathbb{R}$) or bounds($\mathbb{Z}$) complete propagators. Table 1 summarizes how completeness depends on view bijectivity.

Table 1. Completeness of derived propagators

propagator	*view*		
	interval bijective	*interval injective*	*arbitrary*
domain	domain	domain	domain
bounds($\mathcal{D}$)	bounds($\mathcal{D}$)	bounds($\mathcal{D}$)	weakly
bounds($\mathbb{Z}$)	bounds($\mathbb{Z}$)	bounds($\mathbb{R}$)	weakly
bounds($\mathbb{R}$)	bounds($\mathbb{R}$)	bounds($\mathbb{R}$)	weakly

The views for integer variables presented at the beginning of this section have the following properties: minus and offset views are interval bijective, whereas a scale view for $a \in \mathbb{Z}$ on x is always interval injective and only interval bijective if $a = 1$ or $a = -1$ (in which cases it coincides with the identity view or a minus view, respectively). An important consequence is that a bounds($\mathbb{Z}$) complete propagator for the constraint $\sum_i x_i = c$, when instantiated with scale views for the x_i, results in a bounds($\mathbb{R}$) complete propagator for $\sum_i a_i x_i = c$.

Transformation. Like the negation view for Boolean variables, minus views for integer variables 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($\mathbb{Z}$) consistent propagator for multiplication $x \times y = z$. 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 not very efficient. 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 implemented to be idempotent (see Section 7). Implementing three different propagators (all variables strictly positive, x or y strictly positive, only z strictly positive) seems excessive. Here, a single propagator assuming that all views are positive is sufficient, the others can be derived using minus views.

Generalization. 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 than a generalized version. The possibility to use the specialized version when the full power of the general version is not required may save space and time during execution.

The propagator for a linear equality constraint $\sum_{i=1}^{n} x_i = c$ is efficient 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 (This of course also holds true for linear inequality and disequality rather than equality). Similarly, a propagator for alldifferent(x_i) can be generalized to alldifferent($c_i + x_i$) by using offset views for $c_i \in \mathbb{Z}$ on x_i. Likewise, a propagator for the element constraint $\langle c_1, \ldots, c_n \rangle [x] = y$ can be generalized to $\langle c_1, \ldots, c_n \rangle [x + o] = y$ with an offset view, where $o \in \mathbb{Z}$ provides a useful offset for the index variable x. It is important to recall that propagators are derived: in Gecode, the above generalizations are applied to domain as well as bounds complete propagators.

Specialization. We employ *constant views* to specialize propagators. A constant view behaves like a fixed variable. In practice, specialization has two advantages: Fewer variables are needed, which means less space consumption. And specialized propagators can be compiled to more efficient code, if 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 Boolean propagator for $x \wedge y \leftrightarrow 1$ from $x \wedge y \leftrightarrow z$ and constant 1 for z; a propagator for the element constraint $\langle c_1, \ldots, c_n \rangle [y] = z$ derived from a propagator for $\langle x_1, \ldots, x_n \rangle [y] = z$; a reified propagator for $(x = c) \leftrightarrow b$ from $(x = y) \leftrightarrow b$ and a constant c for y; a propagator for counting $|\{i \mid x_i = y\}| = c$ from a propagator for $|\{i \mid x_i = y\}| = z$; and many more.

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

$$\varphi^{-}(c) = \{a_{|\,\mathit{Var}} \mid a \in c\}$$
$$\varphi(c) = \{a[k/z] \mid a \in c\}$$

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

Indexicals. Views that perform arithmetic transformations are related to indexicals [3, 13]. An indexical is a propagator that prunes a single variable and is defined in terms of range expressions. A view is similar to an indexical with a single input variable. However, views are not used to build propagators directly, but to derive new propagators from existing ones. Allowing the full expressivity of indexicals for views would imply giving up our completeness results.

Another related concept are arithmetic expressions, which can be used for modeling in many systems (such as ILOG Solver [10]). In contrast to views, these expressions are not used for propagation directly and, like indexicals, yield no completeness guarantees.

6 Set Variables: Channeling

Set constraints deal with variables whose domains are sets of finite sets. This powerset lattice is a Boolean algebra, so typical constraints are constructed from the Boolean primitives disjunction (union), conjunction (intersection), and negation (complement), and the relations equality and implication (subset).

Transformation and Specialization. As for Boolean and integer variables, views on set variables enable transformation and specialization. Using *complement views* (analogous to Boolean negation) 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 gives us $x \setminus y = z$. Constant views like the empty set or the universe enable specialization; for example, $x \cap y = z$ implements set disjointness if z is the constant empty set.

Channeling views. A channeling view changes the type of the values that a variable can take. Our model already accommodates for this as a view φ_x maps elements between different sets Val and Val'.

An important channeling view is a *singleton view* on an integer variable x, defined as $\varphi_x(v) = \{v\}$. It presents an integer variable as a singleton set variable. Many useful constraints involve both integer and set variables, and some of them can be expressed with singleton views. The simplest constraint is $x \in y$, where x is an integer variable and y a set variable. Singleton views let us implement it as $\{x\} \subseteq y$, and just as easily give us the negated and reified variants. Obviously, this extends to $\{x\} \diamond y$ for all other set relations $\diamond$.

Singleton views can also be used to derive pure integer constraints from set propagators. For example, the constraint $\mathrm{same}([x_1,\ldots,x_n],[y_1,\ldots,y_m])$ states that the two sequences of integer variables take the same values. With singleton views, $\bigcup_{i=1}^{n}\{x_i\} = \bigcup_{j=1}^{m}\{y_j\}$ implements this constraint.

Channeling between domain implementations. Most systems approximate finite set domains as convex sets defined by a lower and an upper bound [7]. However, Hawkins et al. [8] introduced a complete representation for the domains of finite set variables using ROBDDs. Channeling views can translate between 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.

7 Extended Properties of Derived Propagators

This section discusses how views can be composed, how derived propagators behave with respect to idempotence and subsumption, and how events can be used to schedule derived propagators. Finally, we discuss the relation between views and path consistency.

Composing views. A derived propagator permits further derivation: $\widehat{\varphi}(\widehat{\varphi'}(p))$ for two views φ, φ' is perfectly acceptable, properties like correctness and completeness carry over. For instance, we can derive a propagator for $x - y = c$ from a propagator for $x + y = 0$ by combining an offset view and a minus view on y.

Idempotent propagators. A propagator is idempotent iff $p(p(d)) = p(d)$ for all domains d. Some systems require all propagators to be idempotent, others apply optimizations if the idempotence of a propagator is known [11]. If a propagator is derived from an idempotent propagator, the result is idempotent again:

Theorem 6. *If $p(p(d)) = p(d)$ for a propagator p and a domain d, then, for any view φ, $\widehat{\varphi}(p)(\widehat{\varphi}(p)(d)) = \widehat{\varphi}(p)(d)$.* □

Proof. Function composition is associative, so we can write $\widehat{\varphi}(p)(\widehat{\varphi}(p)(d))$ as $\varphi^- \circ p \circ (\varphi \circ \varphi^-) \circ p \circ \varphi(d)$. We know that $\varphi \circ \varphi^- = \mathrm{id}$ for all domains that contain only assignments on which φ^- is fully defined, meaning that $|\varphi^-(d)| = |d|$. As we first apply φ, this is the case here, so we can remove $\varphi \circ \varphi^-$, leaving $\varphi^- \circ p \circ p \circ \varphi(d)$. As p is idempotent, this is equivalent to $\varphi^- \circ p \circ \varphi(d) = \widehat{\varphi}(p)(d)$. ■

Subsumption. A propagator is subsumed for a domain d iff for all stronger domains $d' \subseteq d$, $p(d') = d'$. Subsumed propagators do not contribute any propagation in the remaining subtree of the search, and can therefore be removed. Deciding subsumption is coNP-complete in general, but for most propagators an approximation can be decided easily. This can be used to optimize propagation.

Theorem 7. *p is subsumed by $\varphi(d)$ iff $\widehat{\varphi}(p)$ is subsumed by d.* □

Proof. The definition of φ gives us that $\forall d' \subseteq d.\ \varphi^{-}(p(\varphi(d'))) = d'$ is equivalent with $\forall d' \subseteq d.\ \varphi^{-}(p(\varphi(d'))) = \varphi^{-}(\varphi(d')$. As φ^{-} is a function, and because it is contraction-preserving (see Theorem 3), this is equivalent with $\forall d' \subseteq d.\ p(\varphi(d')) = \varphi(d')$. Because all $\varphi(d')$ are subsets of $\varphi(d)$, we can rewrite this to $\forall d'' \subseteq \varphi(d).\ p(d'') = d''$, concluding the proof. ■

Events. Many systems control propagator invocation using *events* (for a detailed discussion, see [11]). An event describes how a domain changed. Typical events for finite domain integer variables are: the variable x becomes fixed ($\text{fix}(x)$); the lower bound of variable x changes ($\text{lbc}(x)$); the upper bound of variable x changes ($\text{ubc}(x)$); the domain of variable x changes ($\text{dmc}(x)$). In some systems, $\text{lbc}(x)$ and $\text{ubc}(x)$ are collapsed into one event, $\text{bc}(x) = \text{lbc}(x) \vee \text{ubc}(x)$. Events are monotone: if $\text{events}(d, d'')$ is the set of events occurring when the domain changes from d to d'' (with $d'' \subseteq d$), then we have $\text{events}(d, d'') = \text{events}(d, d') \cup \text{events}(d', d'')$ for any $d'' \subseteq d' \subseteq d$. Propagators are associated with *event sets*: A propagator p depends on an event set $es(p)$ iff

1. for all d if $p(d) \neq p(p(d))$, then $\text{events}(d, p(d)) \cap es(p) \neq \emptyset$
2. for all d, d' where $p(d) = d$, $d' \subseteq d$, $p(d') \neq d'$, then $\text{events}(d, d') \cap es(p) \neq \emptyset$

If a propagator p depends on $es(p)$, what event set does $\widehat{\varphi}(p)$ depend on? We can construct a safe approximation of $es(\widehat{\varphi}(p))$: If $\text{fix}(x) \in es(p)$, put $\text{fix}(x) \in es(\widehat{\varphi}(p))$. For any other event $e \in es(p)$, put $\text{dmc}(x) \in es(\widehat{\varphi}(p))$. This is correct because φ_x is injective. If φ_x is monotone with respect to the order on Val_x, $a < b \Rightarrow \varphi_x(a) < \varphi_x(b)$, we can also use bounds events. If φ_x is anti-monotone with respect to that order, we have to switch lbc with ubc.

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

Example. Assume the equality constraint $c \equiv (x = y)$. In order to propagate $c' \equiv (x = y + 1)$, we could use a domain complete propagator p for c and a view φ with $\varphi_x(v) = v$, $\varphi_y(v) = v + 1$. The alternative model would be defined with additional variables x' and y', a view constraint $c_{\varphi,x}$ for $x' = x$, a view constraint $c_{\varphi,y}$ for $y' - 1 = y$, and $c[x/x', y/y']$, yielding $x' = y'$. □

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, and we can reach a fixpoint 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. In this sense, views can be seen as a way for specifying a *perfect order of propagation*, which is usually not possible in constraint programming systems.

If $\widehat{\varphi}(p)$ is domain complete for $\varphi^{-}(c)$, then it achieves *path consistency* for $c[x_1/x'_1, \ldots, x_n/x'_n]$ and all the $c_{\varphi,i}$ in the decomposition model.

8 Limitations

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

Beyond injective views. Views as defined in this paper are required to be injective. This excludes some interesting views, such as a view for the absolute value of a variable, or a view of a variable modulo some constant. None of the basic proofs makes use of injectivity, so non-injective views can be used to derive (bounds) complete, correct propagators.

However, event handling changes when views are not injective:

- A domain change event on a variable does not necessarily translate to a domain change event on the view. For instance, given a domain d with $d(x) = \{-1, 0, 1\}$, removing the value -1 from x is a domain change event on x, but not on $\mathrm{abs}(x)$.
- A domain change event on a variable may result in a value event on the view. For instance, removing 0 instead of -1 in the above example results in $d(x) = \{-1, 1\}$, but in $\mathrm{abs}(x)$ there is only a single value left.

These effects may lead to unnecessary propagtor invocations, or even to incorrect behavior if a propagator relies on the accuracy of the reported event. As propagators in Gecode may assume that events are crisp in this sense, we decided not to allow 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 actual domain. As domains can only represent cartesian products, this is not possible in general. For views that do not preserve contraction, Theorem 7 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. In Gecode, propagators report whether they are subsumed, so that they are not considered for propagation again. This optimization is vital for performance, so we only allow contraction-preserving views.

For contraction-preserving views on multiple variables, all our theorems still hold. Some useful views we could identify are

- A set view of Boolean variables $[b_1, \ldots, b_n]$, behaving like $\{i \mid b_i = 1\}$.
- An integer view of Boolean variables $[b_1, \ldots, b_n]$, where b_i is 1 iff the integer has value i.
- The inverse views of the two views above.

These views are of limited use, and the decomposition approach will probably work just as well in these cases.

Table 2. Applicability of views: number of generic vs. derived propagators

Variable type	*Generic propagators*	*Derived propagators*	*Ratio*
Integer	69	230	3.34
Boolean	23	72	3.13
Set	24	114	4.75
Overall	116	416	3.59

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

For example, a propagator for interval-based finite set variables can assume that adjusting the lower bound of a variable does not affect its upper bound. If this propagator is instantiated with a channeling view for an ROBDD-based set variable, this invariant is violated: if, for instance, the current domain is $\{\{1,2\},\{3\}\}$, and you add 1 to the lower bound, the 3 is removed from the upper bound (in addition to 2 being added to the lower bound). A propagator that relies on the invariant may lose idempotence.

9 Experiments

Our experiments in [12] showed that deriving propagators using views incurs no runtime overhead. Here, we present empirical evidence for two more facts: views are highly applicable in real-world constraint programming systems, and they are clearly superior to a decomposition-based approach.

Applicability. The Gecode C++ library [5] makes heavy use of views. Table 2 shows the number of generic propagators implemented in Gecode, and the number of derived instances. On average, every generic propagator results in 3.59 propagator instances. Propagators in Gecode account for more than 40 000 lines of code and documentation. As a rough estimate, generic propagators with views save around 100 000 lines of code and documentation to be written, tested, and maintained. On the other hand, the views are implemented in less than 8 000 lines of code, yielding a 1250% return on investment.

Views vs. decomposition. In order to relate derived propagators to arc and path consistency, Sect. 7 decomposed a derived propagator $\widehat{\varphi}(p)$ into additional variables and propagators for the individual φ_x and p. Of course, one has to ask why we advertise variable views instead of always using decomposition. Table 3 shows the runtime and space requirements of several benchmarks implemented in Gecode. The numbers were obtained on a Intel Pentium IV at 2.8 GHz running Linux and Gecode 2.1.1. The figures illustrate that derived propagators clearly outperform the decomposition, both in runtime and space.

Table 3. Runtime and space comparison: derived propagators vs. decomposition

Benchmark	*derived*		*decomposed*	
	time (ms)	*space (kB)*	*relative time (%)*	*relative space (%)*
Alpha	91.25	83.22	405.62	167.32
Eq-20	1.37	70.03	613.61	219.95
Queens 100	24.72	2 110.00	705.10	103.03
Golf 8-4-9	310.40	10 502.00	211.47	231.64
Steiner triples 9	135.72	957.03	108.38	100.03

10 Conclusion and Future Work

The paper has developed variable views as a technique to derive perfect propagator variants. Such variants are ubiquitous, and the paper has shown how to systematically derive propagators using techniques such as transformation, generalization, specialization, and channeling.

We have presented a model of views that allowed us to prove that derived propagators are indeed perfect: they inherit correctness and domain completeness from their original propagator, and preserve bounds completeness given additional properties of views.

As witnessed by the empirical evaluation, deriving propagators saves huge amounts of code to be written and maintained in practice, and is clearly superior to decomposing constraints into additional variables and simple propagators.

For future work, it will be interesting to investigate how views can be generalized, even if that means that derived propagators are not perfect any more.

Acknowledgements. We thank Mikael Lagerkvist and Gert Smolka for fruitful discussions about views and helpful comments on a draft of this paper.

References

1. K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96)*, volume 1139 of *LNCS*, pages 62–76. Springer, 1996.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97*, volume 1292 of *LNCS*, pages 191–206, Southampton, UK, 1997. Springer.
4. C. W. Choi, W. Harvey, J. H. M. Lee, and P. J. Stuckey. Finite domain bounds consistency revisited. In A. Sattar and B.-H. Kang, editors, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *LNCS*, pages 49–58. Springer, 2006.
5. Gecode: Generic constraint development environment, 2008. Available as an open-source library from `www.gecode.org`.
6. I. P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In F. Benhamou, editor, *Twelfth International Conference on Principles*

and Practice of Constraint Programming, volume 4204 of *LNCS*, pages 182–197, Nantes, France, 2006. Springer.
7. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
8. P. Hawkins, V. Lagoon, and P. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR)*, 24:109–156, 2005.
9. M. J. Maher. Propagation completeness of reactive constraints. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, volume 2401 of *LNCS*, pages 148–162, London, UK, 2002. Springer.
10. J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 513–527, Portland, OR, USA, Dec. 1995. The MIT Press.
11. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008. To appear.
12. C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *LNAI*, pages 118–132. Springer, 2006.
13. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, Oct. 1998.