Views and Iterators for Generic Constraint Implementations

Christian Schulte[1] and Guido Tack[2]

[1] ICT, KTH - Royal Institute of Technology, Sweden, schulte@imit.kth.se
[2] PS Lab, Saarland University, Saarbrücken, Germany, tack@ps.uni-sb.de

Abstract. This paper introduces an architecture for generic constraint implementations based on variable views and range iterators. Views allow, for example, to scale, translate, and negate variables. The paper shows how to make constraint implementations generic and how to reuse a single generic implementation with different views for different constraints. A wide range of applications of views exemplifies their usefulness and their potential for simplifying constraint implementations. We introduce domain operations compatible with views based on range iterators. The paper evaluates the applicability of the approach as well as different implementation techniques for the presented architecture.

1 Introduction

A challenging aspect in developing and extending a constraint programming system is implementing a *comprehensive* set of constraints. Ideally, a system should provide simple, expressive, and efficient abstractions that ease development and reuse of constraint implementations.

This paper contributes a new architecture based on variable views and range iterators. The architecture comprises an additional level of abstraction to decouple variable implementations from constraint implementations, the propagators. Propagators compute generically with variable views instead of variables.

A view of a variable presents an adaptor that performs transformations while accessing the variable it abstracts over. Views support operations like scaling, translation, and negation of variables. Views also abstract over the underlying data structure used for storing the variable domain. That way, cross-domain views can for example enable propagators for finite set constraints to operate on finite domain variables.

This simple layer of abstraction allows one propagator to be instantiated multiple times, with different views. For example, a simple generic propagator for linear equality $\sum_{i=1}^{k} x_i = c$ can be used with a scale-view $x_i = a_i \cdot y_i$ to obtain an implementation of $\sum_{i=1}^{k} a_i \cdot y_i = c$. Or a negated Boolean view can be used to derive an implementation of Boolean disjunction from a propagator for conjunction. As a final example, a cross-domain view of a finite domain variable as a singleton set, together with a subset propagator, yields a propagator for $x \in s$. Variable views thus assist in implementing propagators on a higher level of abstraction.

Range iterators support powerful and efficient domain operations on variables and variable views. The operations can access and modify multiple values of a variable

domain simultaneously. Range iterators are efficient as they help avoiding temporary data structures. They simplify propagators by serving as adaptors between variables and propagator data structures.

The architecture is carefully separated from its implementation. Two different implementation approaches are presented and evaluated. An implementation using parametric polymorphism (such as templates in C++) is shown to not incur any runtime cost. The architecture can be used for arbitrary constraint programming systems and has been fully implemented in Gecode [2].

Plan of the paper. The next section presents a model for finite domain constraint programming systems. Sect. 3 introduces variable views and exemplifies their use. Sect. 4 presents Boolean views of finite domain variables and discusses pairs of symmetric propagators. Sect. 5 introduces iterator-based domain operations that are applied to views in the following section. Variable views for set constraints are discussed in Sect. 7. In Sect. 8 implementation approaches for views and iterators are presented, followed by their evaluation in Sect. 9. The last section concludes and discusses future work.

2 Constraint Programming Systems

This section introduces the model for finite domain constraint programming systems considered in this paper and relates it to existing systems.

Variables and propagators. Finite domain constraint programming systems offer services to support constraint propagation and search. In this paper we are only concerned with variables used for constraint propagation. We assume that a constraint is implemented by a *propagator*. A propagator maintains a collection of variables and performs constraint propagation by executing operations on them. In the following we consider finite domain variables and propagators. A finite domain variable x has an associated *domain* $\mathrm{dom}(x)$ being a subset of some finite subset of the integers.

Propagators do not manipulate variable domains directly but use operations provided by the variable. These operations return information about the domain or update the domain. In addition, they handle failure (the domain becomes empty) and control propagation.

Value operations. A *value operation* on a variable involves a single integer as result or argument. We assume that a variable x with $D = \mathrm{dom}(x)$ provides the following value operations: $x.\texttt{getmin}()$ returns $\min D$, $x.\texttt{getmax}()$ returns $\max D$, $x.\texttt{adjmin}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \geq n\}$, $x.\texttt{adjmax}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \leq n\}$, and $x.\texttt{excval}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \neq n\}$. These operations are typical for finite domain constraint programming systems like Choco [6], ILOG Solver [9, 11, 4], Eclipse [1], Mozart [8], and Sicstus [5]. Some systems provide additional operations such as for assigning values.

Domain operations. A *domain operation* supports simultaneous access or update of multiple values of a variable domain. In many systems this is provided by supporting an abstract set-datatype for variable domains, as for example in Choco [6], Eclipse [1], Mozart [8], and Sicstus [5]. ILOG Solver [9, 11, 4] only allows access by iterating over the values of a variable domain.

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

3 Variable Views with Value Operations

This section introduces variable views with value operations. The full design with domain operations and a discussion of their properties follows in Sect. 6.

Example 1 (Smart n-Queens). Consider the well-known finite domain constraint model for n-Queens using three alldifferent constraints: each queen is represented by a variable x_i ($0 \leq i < n$) with domain $\{0, \ldots, n-1\}$. The constraints state that the values of all x_i, the values of all $x_i - i$, and the values of all $x_i + i$ must be pairwise different for $0 \leq i < n$.

If the used constraint programming system lacks versions of alldifferent supporting that the values of $x_i + c_i$ are different, the user must resort to using additional variables y_i and constraints $y_i = x_i + c_i$ and the single constraint that the y_i are different. This approach is clearly not very efficient: it triples the number of variables and requires additional $2n$ binary constraints.

Systems with this extension of alldifferent must implement two very similar versions of the same propagator. This is tedious and increases the amount of code that requires maintenance. In the following we make propagators *generic*: the same propagator can be reused for several variants.

To make a propagator generic, all its operations on variables are replaced by operations on variable views. A *variable view* (view for short) implements the same operations as a variable. A view stores a reference to a variable. Invoking an operation on the view executes the appropriate operation on the view's variable. Multiple variants of a propagator can be obtained by instantiating the single generic propagator with multiple different variable views.

Offset-views. For an *offset-view* $v = \text{voffset}(x, c)$ for a variable x and an integer c, performing an operation on v results in performing an operation on $x + c$. The operations on the offset-view are:

$$
\begin{array}{ll}
v.\texttt{getmin}() := x.\texttt{getmin}() + c & v.\texttt{getmax}() := x.\texttt{getmax}() + c \\
v.\texttt{adjmin}(n) := x.\texttt{adjmin}(n - c) & v.\texttt{adjmax}(n) := x.\texttt{adjmax}(n - c) \\
v.\texttt{excval}(n) := x.\texttt{excval}(n - c) &
\end{array}
$$

To obtain both alldifferent propagators required by Example 1, also an *identity-view* is needed. An operation on an identity-view $\mathrm{vid}(x)$ for a variable x performs the same operation on x. That is, identity-views turn variables into views to comply with propagators now computing with views. In an implementation language that supports subtyping, variables can themselves be regarded as views, eliminating the need for identity views.

Obtaining the two variants of alldifferent is straightforward: the propagator is made generic with respect to which view it uses. Using the propagator with both an identity-view and an offset-view yields the required propagators.

Offset-views can also be used to obtain propagators for strict inequalities from propagators for the non-strict constraints. For instance, $x < y$ can be implemented as $x \leq \mathrm{voffset}(y,-1)$.

Sect. 8 discusses how views can be implemented whereas this section focuses on the architecture only. However, to give some intuition, in C++ for example, propagators can be made generic by implementing them as templates with the used view as template argument. Instantiating the generic propagator then amounts to instantiating the corresponding template with a particular view.

Views are orthogonal to the propagator. In the above example, offset-views can be used for any implementation of alldifferent using value operations. This includes the naive version propagating when variables become assigned or the bounds-consistent version [10].

Scale-views. In the above example, views allow to reuse the same propagator for variants of a constraint, avoiding duplication of code and effort. In the following, views can also simplify the implementation of propagators.

Example 2 (Linear inequalities). A common constraint is linear inequality $\sum_{i=1}^{n} a_i \cdot x_i \leq c$ (equality and disequality is similar) with integers a_i and c and variables x_i. In the following we restrict the a_i to be positive.

A typical bounds-propagator executes for $1 \leq j \leq n$:

$$x_j.\mathtt{adjmax}(\lfloor (c-l_j)/a_j \rfloor) \quad \text{with} \quad l_j = \textstyle\sum_{i=1,i\neq j}^{n} a_i \cdot x_i.\mathtt{getmin}()$$

Quite often, models feature the special case $a_i = 1$ for $1 \leq i \leq n$. For this case, it is sufficient to execute for $1 \leq j \leq n$:

$$x_j.\mathtt{adjmax}(c-l_j) \quad \text{with} \quad l_j = \textstyle\sum_{i=1,i\neq j}^{n} x_i.\mathtt{getmin}()$$

As this case is common, a system should optimize it. An optimized version requires less space (no a_i required) and less time (no multiplication, division, and rounding). But, a more interesting question is: can one just implement the simple propagator and get the full version by using views?

With scale-views, the simple implementation can be used in both cases. A *scale-view* $v = \mathrm{vscale}(a,x)$ for a positive integer $a > 0$ and a variable x defines operations for $a \cdot x$:

$$\begin{array}{ll} v.\mathtt{getmin}() := a \cdot x.\mathtt{getmin}() & v.\mathtt{getmax}() := a \cdot x.\mathtt{getmax}() \\ v.\mathtt{adjmin}(n) := x.\mathtt{adjmin}(\lceil n/a \rceil) & v.\mathtt{adjmax}(n) := x.\mathtt{adjmax}(\lfloor n/a \rfloor) \\ v.\mathtt{excval}(n) := \textbf{if } n \bmod a = 0 \textbf{ then } x.\mathtt{excval}(n/a) & \end{array}$$

From the simpler implementation the special case (identity-views) and the general case (scale-views) can be obtained. Multiplication, division, and rounding is separated from actually propagating the inequality constraint. Views hence support separation of concerns and can simplify the implementation of propagators. In particular, multiplication, division, and rounding need to be implemented only once for the scale-view: any generic propagator can use scale-views.

Minus-views. Another common optimization is to implement binary and ternary variants of commonly used constraints. This optimization reduces the overhead with respect to both time and memory as no array is needed.

Example 3 (Binary linear inequality). Consider a propagator for $v_1 + v_2 \leq c$ with views v_1 and v_2 propagating as described in Example 2. With scale-views $v_1 = \text{vscale}(a_1, x_1)$ and $v_2 = \text{vscale}(a_2, x_2)$ the propagator also implements $a_1 \cdot x_1 + a_2 \cdot x_2 \leq c$ provided that $a_1, a_2 > 0$. However, $x_1 - x_2 \leq c$ cannot be obtained with scale-views. Even if scale-views allowed negative constants, it would be inefficient to multiply, divide, and round to just achieve negation.

A *minus-view* $v = \text{vminus}(x)$ for a variable x provides operations such that v behaves as $-x$. Its operations reflect that the smallest possible value for x is the largest possible value for $-x$ and vice versa:

$$\begin{aligned} v.\texttt{getmin}() &:= -x.\texttt{getmax}() & v.\texttt{getmax}() &:= -x.\texttt{getmin}() \\ v.\texttt{adjmin}(n) &:= x.\texttt{adjmax}(-n) & v.\texttt{adjmax}(n) &:= x.\texttt{adjmin}(-n) \\ v.\texttt{excval}(n) &:= x.\texttt{excval}(-n) \end{aligned}$$

With minus-views, $x_1 - x_2 \leq c$ can be obtained from an implementation of $v_1 + v_2 \leq c$ with $v_1 = \text{vid}(x_1)$ and $v_2 = \text{vminus}(x_2)$. With an offset-view it is actually sufficient to implement $v_1 + v_2 \leq 0$. Then $x_1 + x_2 \leq c$ can be implemented by an identity-view $\text{vid}(x_1)$ for v_1 and an offset-view $\text{voffset}(x_2, -c)$ for v_2. But again, given just $v_1 + v_2 \leq 0$, an implementation for $x_1 - x_2 \leq c$ with $c \neq 0$ cannot be obtained.

Minus-views implement the inverse for finite domain variables, thus all propagators that are symmetric with respect to the sign of their arguments can take advantage of minus views. An example for a pair of symmetric propagators on finite domain variables is minimum and maximum: $\max(x_1, \ldots, x_n)$ can be obtained from a the minimum propagator with $\min(\text{vminus}(x_1), \ldots, \text{vminus}(x_n))$. We will come back to inverse views in the sections about Boolean and set constraints.

Derived views. It is unnecessarily restrictive to define views in terms of variables. The actual requirement for a view is that its variable provides the same operations. It is straightforward to make views generic themselves: views can be defined in terms of other views. The only exception are identity-views as they serve the very purpose of casting a variable into a view. Views such as offset, scale, and minus are called *derived views*: they are derived from some other view.

With derived views being defined in terms of views, the first step to use a derived view is to turn a variable into a view by an identity-view. For example, a minus-view v for the variable x is obtained from a minus-view and an identity-view: $v = \text{vminus}(\text{vid}(x))$.

Example 4 (Binary linear inequality reconsidered). Using offset-views, minus-views, and scale-views, all possible variants of binary linear inequalities can now be obtained from a propagator for $v_1 + v_2 \leq 0$. For example, $a \cdot x_1 - x_2 \leq c$ with $a > 0$ can be obtained with $v_1 = \text{vscale}(a, \text{vid}(x_1))$ and $v_2 = \text{vminus}(\text{voffset}(\text{vid}(x_2), c))$ or $v_2 = \text{voffset}(\text{vminus}(\text{vid}(x_2)), -c)$.

Scale-views reconsidered. The coefficient of a scale-view is restricted to be positive. Allowing arbitrary non-zero constants a in a scale-view $s = \text{vscale}(a, x)$ requires to take the signedness of a into account. This can be seen for the following two operations (the others are similar):

$$s.\texttt{getmin}() := \textbf{if } a < 0 \textbf{ then } a \cdot x.\texttt{getmax}() \textbf{ else } a \cdot x.\texttt{getmin}()$$
$$s.\texttt{adjmax}(n) := \textbf{if } a < 0 \textbf{ then } x.\texttt{adjmin}(\lceil n/a \rceil) \textbf{ else } x.\texttt{adjmax}(\lfloor n/a \rfloor)$$

This extension might be inefficient. Consider Example 2: inside the loop implementing propagation on all views, the decision whether the coefficient in question is positive or negative must be made. For modern computers, conditionals — in particular in tight loops — can reduce performance considerably. A more efficient way is to restrict scale-views to positive coefficients and use an additional minus-view for cases where negative coefficients are required.

Example 5 (Linear inequalities reconsidered). An efficient way to implement a propagator for linear inequality distinguishes positive and negative variables as in $\sum_{i=1}^{n} x_i + \sum_{i=1}^{m} -y_i \leq c$.

The propagator is simple: it consists of two parts, one for the x_i and one for the y_i. Both parts share the same implementation used with different views. To propagate to the x_i, identity-views are used. To propagate to the y_i, minus-views are used. Arbitrary coefficients are obtained from scale-views as shown above.

The example shows that it can be useful to make parts of a propagator generic and reuse these parts with different views. Puget presents in [10] an algorithm for the bounds-consistent alldifferent. The paper presents only an algorithm for adjusting the upper bounds of the variables x_i and states that the lower bounds can be adjusted by using the same algorithm on variables y_i where $y_i = -x_i$. With views, this technique for simplifying the presentation of an algorithm readily carries over to its implementation: the implementation can be reused together with minus-views.

Constant-views. Derived views exploit that views do not need to be implemented in terms of variables. This can be taken to the extreme in that a view has no access at all to a variable. A constant-view $v = \text{vcon}(c)$ for an integer c provides operations such that v behaves as a variable x being equal to c:

$$v.\texttt{getmin}() := c \qquad v.\texttt{getmax}() := c$$
$$v.\texttt{adjmin}(n) := \textbf{if } n > c \textbf{ then fail} \qquad v.\texttt{adjmax}(n) := \textbf{if } n < c \textbf{ then fail}$$
$$v.\texttt{excval}(n) := \textbf{if } n = c \textbf{ then fail}$$

Example 6 (Ternary linear inequalities). Another optimization for linear constraints are ternary variants. Given a propagator for $v_1 + v_2 + v_3 \leq c$ and using a constant-view $\text{vcon}(0)$ for one of the views v_i, all binary variants as discussed earlier can be obtained.

In summary, for linear inequalities (this carries over to linear equalities and disequalities), views support many optimized special cases from just two implementations (the general n-ary case and the ternary case). These implementations are simple as they do not need to consider coefficients.

4 Boolean Views

Constraints on 0/1 variables are a special case of finite domain constraints. However, specialized propagators can take advantage of the more precise knowledge about the domain.

A *Boolean-view* of a finite domain variable extends the variable's interface with operations for testing its value ($x.\texttt{zero}()$, $x.\texttt{one}()$, $x.\texttt{none}()$) and assigning the variable ($x.\texttt{assign\_one}()$, $x.\texttt{assign\_zero}()$). Propagators specialized for Boolean-views, such as equality ($b_1 = b_2$), conjunction ($(b_1 \wedge b_2) \Leftrightarrow b_3$), and equivalence ($(b_1 = b_2) \Leftrightarrow b_3$), can be implemented in a straightforward way using this interface.

Symmetric Boolean propagators. The inverse of a Boolean is its logical negation, implemented by a *negated Boolean-view*. The operations for a negated Boolean-view $v = \text{vneg}(x)$ are straightforward:

$v.\texttt{zero}()$	$:= x.\texttt{one}()$	$v.\texttt{one}()$	$:= x.\texttt{zero}()$
$v.\texttt{none}()$	$:= x.\texttt{none}()$		
$v.\texttt{assign\_one}()$	$:= x.\texttt{assign\_zero}()$	$v.\texttt{assign\_zero}()$	$:= x.\texttt{assign\_one}()$

Example 7 (Ternary disjunction). Boolean disjunction $(x \vee y) \Leftrightarrow z$ can be implemented as $(\neg x \wedge \neg y) \Leftrightarrow \neg z$. This translates directly to an instance of the Boolean conjunction propagator. Similarly, other Boolean propagators such as exclusive or and implication can be derived.

5 Domain Operations and Range Iterators

Today's constraint programming systems support domain operations either only for access or by means of an explicitly represented abstract datatype. In this paper, we propose domain operations based on range iterators. These operations are shown to be simple, expressive, and efficient. Additionally, range iterators are essential for views as presented in Sect. 6.

Range iterators. A *range iterator* r for a range sequence $s = \langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ allows to iterate over s: each of the $[n_i \,..\, m_i]$ 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 possible implementation of a range iterator r for s maintains an index i_r which is initially $i_r = 1$, the operations can then be defined as:

$$r.\texttt{done}() := i_r > k \qquad r.\texttt{next}() := (i_r \leftarrow i_r + 1)$$
$$r.\texttt{min}() := n_{i_r} \qquad r.\texttt{max}() := m_{i_r}$$

A range iterator hides its implementation. Iteration can be by position as above, but it can also be by traversing a list. The latter is particularly interesting if variable domains are implemented as lists of ranges themselves.

Iterators are consumed by iteration. Hence, if the same sequence needs to be iterated twice, a fresh iterator is needed. If iteration is cheap, a reset-operation for an iterator can be provided so that multiple iterations are supported by the same iterator. For more expensive iterators, a solution is discussed later.

Domain operations. Variables are extended with operations to access and modify their domains with range iterators. For a variable x, the operation $x.\texttt{getdom}()$ returns a range iterator for ranges$(\text{dom}(x))$. For a range iterator r the operation $x.\texttt{setdom}(r)$ updates $\text{dom}(x)$ to $\text{set}(r)$ provided that $\text{set}(r) \subseteq \text{dom}(x)$. The responsibility for ensuring that $\text{set}(r) \subseteq \text{dom}(x)$ is left to the programmer and hence requires careful consideration. Later richer (and safe) domain operations are introduced. The operation $x.\texttt{setdom}(r)$ is *generic* with respect to r: any range iterator can be used.

Domain operations can offer a substantial improvement over value operations, if many values need to be removed from a variable domain simultaneously. Assume a typical implementation of a variable domain D which organizes ranges$(D) = \langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ as a linked-list. Removing a single element from D takes $O(k)$ time and might increase the length of the linked-list by one (introducing an additional hole). Hence, in the worst case, removing l elements takes $O(l(k+l))$ time. With domain operations based on iterators, removal takes $O(k+l)$ time, as the update can be implemented as one linear pass over the linked list.

Range iterators serve as simplistic abstract datatype to describe finite sets of integers. However, they provide some essential advantages over an explicit set representation. First, any range iterator regardless of its implementation can be used to update the domain of a variable. This turns out to allow for simple, efficient, and expressive updates of variable domains. Second, no costly memory management is required to maintain a range iterator as it provides access to only one range at a time. Third, iterators are essential in providing domain operations on variable views as will be discussed in Sect. 6.

Intersection iterators. Let us consider intersection as an example for computing with range iterators. Intersection is computed by an intersection iterator $r = \text{iinter}(a,b)$, taking two range iterators a and b as input where $\text{set}(r) = \text{set}(a) \cap \text{set}(b)$. The intersection iterator maintains integers n and m for storing the smallest and largest value of its current range. When initialized, the operation $r.\texttt{next}()$ is executed once. The operations are shown in Figure 1.

The **repeat**-loop iterates a and b until their ranges overlap. The tests whether a or b are done ensure that no operation is performed on a done iterator. The remainder computes the resulting range and prepares for computing a next range.

The iterators a and b can be arbitrary iterators (again, the intersection iterator is *generic*), so it is easy to obtain an iterator that computes the intersection of three iterators by using two intersection iterators. Intersection is but one example for a generic

```
r.done() := a.done() ∨ b.done()
r.min() :=  n
r.max() :=  m
r.next() := if a.done() ∨ b.done() then return
            repeat
               while ¬a.done() ∧ (a.max() < b.min()) do a.next()
               if a.done() then return
               while ¬b.done() ∧ (b.max() < a.min()) do b.next()
               if b.done() then return
            until a.max() ≥ b.min()
            n ← max(a.min(), b.min()); m ← min(a.max(), b.max())
            if a.max() < b.max() then a.next() else b.next()
```

Fig. 1. Operations of an intersection iterator

iterator, other useful iterators are for example: iunion(a,b) for iterating the union of a and b, iminus(a,b) for iterating the set difference of a and b, and icompl(a) for iterating the complement of a with respect to some fixed universe.

Example 8 (Propagating equality). Consider a propagator that implements domain-consistent equality: $x = y$ (assuming that x and y are variables, views are discussed later). The propagator can be implemented as follows: get range iterators for x and y by $rx = x.\texttt{getdom}()$ and $ry = y.\texttt{getdom}()$, create an intersection iterator $ri = \text{iinter}(rx, ry)$, update one of the variable domains by $x.\texttt{setdom}(ri)$, and copy the domain from x to y by $y.\texttt{setdom}(x.\texttt{getdom}())$.

Cache-iterators. The above example suggests that for some propagators it is better to actually create an intermediate representation of the range sequence computed by an iterator. The intermediate representation can be reused as often as needed. This is achieved by a *cache-iterator*: it takes an arbitrary range iterator as input, iterates it completely, and stores the obtained ranges in an array. Its actual operations then use the array. The cache-iterator also implements a reset operation as discussed above. By this, the possibly costly input iterator is used only once, while the cache-iterator can be used as often as needed.

Richer domain operations. With the help of iterators, richer domain operations are effortless. For a variable x and a range iterator r, the operation $x.\texttt{adjdom}(r)$ replaces $\text{dom}(x)$ by $\text{dom}(x) \cap \text{set}(r)$, whereas $x.\texttt{excdom}(r)$ replaces $\text{dom}(x)$ by $\text{dom}(x) \setminus \text{set}(r)$:

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

Value versus range iterators. Another design choice is to base domain operations on value iterators: iterate values rather than ranges of a set. This is not efficient: a value sequence is considerably longer than a range sequence (in particular for the common case of a singleton range sequence).

For implementing propagators, however, it can be simpler to iterate values. This can be achieved by a range-to-value iterator. A value iterator v has the operations $v.\texttt{done}()$, $v.\texttt{next}()$, and $v.\texttt{val}()$ to access the current value. A range-to-value iterator takes a range iterator as input and returns a value iterator iterating the values of the range sequence. The inverse is a value-to-range iterator: it takes as input a value iterator and returns the corresponding range iterator.

Iterators as adaptors. Global constraints are typically implemented by a propagator computing over some involved data structure, such as for example a variable-value graph for domain-consistent alldifferent [12]. After propagation, the new variable domains must be transferred from the data structure to the variables. This can be achieved by using a range or value iterator as adaptor. The adaptor operates on the data structure and iterates the range or value sequence for a particular variable. The iterator then can be passed to the appropriate domain operation.

6 Variable Views with Domain Operations

This section discusses domain operations for variable views using iterators.

Identity and constant views. Domain operations for identity-views and constant-views are straightforward. The domain operations for an identity-view $v = \text{vid}(x)$ use the domain operations on x: $v.\texttt{getdom}() := x.\texttt{getdom}()$ and $v.\texttt{setdom}(r) := x.\texttt{setdom}(r)$. For a constant-view $v = \text{vcon}(c)$, the operation $v.\texttt{getdom}()$ returns an iterator for the singleton range sequence $\langle [c \,..\, c] \rangle$. The operation $v.\texttt{setdom}(r)$ just checks whether the range sequence of r is empty.

Derived views. Domain operations for an offset-view $\text{voffset}(v,c)$ are provided by an offset-iterator. The operations of an offset-iterator o for a range iterator r and an integer c (created by $\text{ioffset}(r,c)$) are as follows:

$$o.\texttt{min}() := r.\texttt{min}() + c \qquad o.\texttt{max}() := r.\texttt{max}() + c$$
$$o.\texttt{done}() := r.\texttt{done}() \qquad o.\texttt{next}() := r.\texttt{next}()$$

The domain operations for an offset view $v = \text{voffset}(x,c)$ are as follows:

$$v.\texttt{getdom}() := \text{ioffset}(x.\texttt{getdom}(), c) \qquad v.\texttt{setdom}(r) := x.\texttt{setdom}(\text{ioffset}(r, -c))$$

For minus-views we just give the range sequence as iteration is obvious. For a given range sequence $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$, the negative sequence is obtained by reversal and sign change as $\langle [-m_{k-i+1} \,..\, -n_{k-i+1}] \rangle_{i=1}^{k}$. The same iterator for this sequence can be used both for `setdom` and `getdom` operations. Note that the iterator is quite complicated as it changes direction of the range sequence, possible implementations are discussed in Sect. 8.

Assume a scale-view $s = \text{vscale}(a,v)$ with $a > 0$ and $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ being a range sequence for v. If $a = 1$, the range sequence remains unchanged. Otherwise, the corresponding range sequence for s is $\langle \{a \cdot n_1\}, \{a \cdot (n_1 + 1)\}, \ldots, \{a \cdot m_1\}, \ldots, \{a \cdot n_k\}, \{a \cdot (n_k + 1)\}, \ldots, \{a \cdot m_k\} \rangle$.

Assume that $\langle [n_i \mathinner{..} m_i] \rangle_{i=1}^{k}$ is a range sequence for s. Then for $1 \leq i \leq k$ the ranges $[\lceil n_i/a \rceil \mathinner{..} \lfloor m_i/a \rfloor]$ correspond to the required variable domain for v, however they do not necessarily form a range sequence as the ranges might be empty, overlapping, or adjacent. Iterating the range sequence is simple by skipping empty ranges and conjoining overlapping or adjacent ranges.

Consistency. An important issue is how views affect the consistency of a propagator. Let us first consider all views except scale-views. These views compute bijections on the values as well as on the ranges of a domain D. A bounds (domain) consistent propagator for a constraint C with variables $x_1, \ldots, x_n$ establishes bounds (domain) consistency for the constraint C with all the variables replaced by $v_k(x_k)$ (if v_k computes the view of x_k).

Scale-views only compute bijections on values: a range does not remain a range after multiplication. This implies that bounds consistent propagators do not establish bounds consistency on scale-views. Consider for example a bounds consistent propagator for alldifferent. With $x, y, z \in \{1, 2\}$, alldifferent$(4x, 4y, 4z)$ cannot detect failure, while alldifferent(x, y, z) can. Note that this is not a limitation of our approach but a property of multiplication.

7 Views for Set Constraints

Views and iterators readily carry over to other constraint domains. This section shows how to apply them to finite sets.

Finite sets. Most systems approximate the domain of a finite set variable by a greatest lower and least upper bound [3]: $\mathrm{dom}(x) = (\mathrm{glb}(x), \mathrm{lub}(x))$. The fundamental operations are similar to domain operations on finite domain variables: $x.\texttt{getglb}()$ returns $\mathrm{glb}(x)$, $x.\texttt{getlub}()$ returns $\mathrm{lub}(x)$, $x.\texttt{adjglb}(D)$ updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x) \cup D, \mathrm{lub}(x))$, and $x.\texttt{adjlub}(D)$ updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x), \mathrm{lub}(x) \cap D)$.

All these operations take sets as arguments or return them. As the abstract datatype we use for representing sets is an iterator, iterators play the central role here. In fact, range iterators provide exactly the operations that set propagators need: union, intersection, and complement. Most propagators thus do not require temporary data structures.

As for finite domain variables, set propagators now operate on set views. The obvious views for set variables are the identity view and constant-views – like the empty set, the universe, or some arbitrary set. Constant-views again help derive binary propagators from ternary ones. For example, $s_1 \cap s_2 = s_3$ implements set disjointness if s_3 is the constant empty set.

Symmetric set constraints. The inverse of a set variable is its complement. A *complement view* $v = \mathrm{vcompl}(x)$ of a set view x can be easily derived using the iterators already introduced:

$$v.\texttt{getglb}() := \mathrm{icompl}(x.\texttt{getlub}()) \qquad v.\texttt{getlub}() := \mathrm{icompl}(x.\texttt{getglb}())$$
$$v.\texttt{adjglb}(D) := x.\texttt{adjlub}(\mathrm{icompl}(D)) \qquad v.\texttt{adjlub}(D) := x.\texttt{adjglb}(\mathrm{icompl}(D))$$

The propagators for symmetric constraints over Boolean views readily carry over to sets: $x_1 = x_2 \cup x_3$ can be implemented as $\mathrm{vcompl}(x_1) = \mathrm{vcompl}(x_2) \cap \mathrm{vcompl}(x_3)$, and $s_1 = s_2 \setminus s_3$ is equivalent to $s_1 = s_2 \cap \mathrm{vcompl}(s_3)$.

Cross-domain views. With finite domain and set constraints in a single system, cross-domain views come into play. The most obvious cross-domain view is a finite domain variable viewed as singleton set. Using generic propagators, this immediately leads to domain-connecting constraints.

Cross-domain views can support more than one implementation for the same variable type. Set variables, for example, can be implemented with lower and upper bounds or with their full domain using ROBDDs [7]. A cross-domain view allows lower/upper bound propagators to operate on ROBDD-based sets, reusing propagators for which no efficient BDD representation exists.

Finite domain constraints from set propagators. Singleton-views can also be used to derive pure finite domain constraints from set propagators. For example, the constraint same$([x_1,\ldots,x_n],[y_1,\ldots,y_m])$ states that the two sequences of finite domain variables take the same values. Using singleton views, $\bigcup_{i=1}^{n}\{x_i\} = \bigcup_{j=1}^{m}\{y_j\}$ yields an implementation for this constraint. If $m = n$, and all variables must take different values, a disjoint union can be used instead.

8 Implementation

The presented architecture can be implemented as an orthogonal layer of abstraction for any constraint programming system. This section presents the fundamental mechanisms necessary for iterators and views.

Polymorphism. The implementation of generic propagators, views, and iterators requires *polymorphism*: propagators operate on different views, domain operations and iterators on different iterators. Both subtype polymorphism (through inheritance in Java, inheritance and virtual methods in C++) and parametric polymorphism (through templates in C++, generics in Java, polymorphic functions in ML or Haskell) can be used.

In C++, parametric polymorphism through templates is resolved at compile-time, and the generated code is monomorphic. This enables the compiler to perform aggressive optimizations, in particular inlining. The hope is that the additional layer of abstraction can be optimized away entirely. Some ML compilers also apply monomorphization, so similar results could be achieved. Java generics are compiled into casts and virtual method calls, any optimization is left to the just-in-time compiler.

Achieving high efficiency in C++ with templates sacrifices expressiveness. Instantiation can *only* happen at compile-time. Hence, either C++ must be used for modeling, or all potentially required propagator variants must be provided by explicit instantiation. The *choice* which propagator to use can however be made at runtime: for linear equations, for instance, we can test if all coefficients are units, or all are positive, and post the respective optimized propagators. In Gecode, we currently only use template-based polymorphism.

For the instantiation of templates as well as for inlining, the code that is instantiated or inlined must be available at compile time of the code that uses it. This is why most of the actual code in Gecode resides in C++ header files, slowing down compilation of the system. On the interface level however, no templates are used, such that the header files needed for *using* the library are reasonably small.

System requirements. Variable views and range iterators can be added as an orthogonal extension to existing systems. While value operations are not critical as discussed in Sect. 2, depending on which domain operations a system provides, efficiency can differ. In the worst case, domain operations need to be translated into value operations. This would decrease efficiency considerably, however intermediate computations on range iterators would still be carried out efficiently.

A particularly challenging aspect is reversal of range sequences required for the minus-iterator. One approach to implement reversal is to extend all iterators such that they can iterate both backwards and forwards. Another approach is similar to a cache-iterator: store the ranges generated from the input iterator in an array and iterate in reverse order from the array. In Gecode, we have chosen so far the latter approach due to its simplicity. We are going to explore also the former approach: as variable domains in Gecode are provided as doubly-linked lists, iteration in both directions can be provided efficiently.

9 Analysis and Evaluation

This section analyzes the impact different implementations of iterators and views have on efficiency. Two aspects are evaluated: compile-time polymorphism versus run-time polymorphism, and iterators versus temporary data structures.

The experiments use the Gecode C++ (version 1.0.0) constraint programming library [2]. All tests were carried out on a Intel Pentium IV with 2.8GHz and 1GB of RAM, using Linux and the GNU C++ compiler, version 3.4.3. Runtimes are the average of 20 runs, with a coefficient of deviation less than 2% for all benchmarks. Gecode is competitive in efficiency with state-of-the art systems, a comparison is available on the Gecode web pages [2].

The *optimized* column in Table 1 gives the time in milliseconds of the optimized system, the other columns are relative to *optimized*. The examples used are standard benchmarks, the first group using only finite domain constraints, the second group using mainly set constraints.

Code inspection. A thorough inspection of the code generated by the GNU C++ compiler and the Microsoft Visual C++ compiler shows that they actually perform the optimizations we consider essential. Operations on both views and iterators are inlined entirely and thus implemented in the most efficient way. The abstractions do not impose a run-time penalty (compared to a system without views and iterators).

Templates versus virtual methods. As the previous section suggested, in C++, compile-time polymorphism using templates is far more efficient than virtual method calls. To evaluate this, we changed the basic operations of finite domain views such that they cannot be inlined. The required changes are rather involved, so we did not try the same for iterators and set views. An implementation based on virtual methods will typically exhibit an even higher overhead. Table 1 shows the results in column *no-inline*. Function calls that are not inlined cause a runtime overhead between 29% and 58%.

Table 1. Runtime comparison

Benchmark	*optimized*	*no-inline*	*temporary*
	time in ms	*relative %*	
Alpha	122.85	141.30	103.70
Donald	0.64	155.60	114.70
Golomb 10 (bound)	1 260.50	158.20	101.10
Golomb 10 (domain)	2 064.00	129.70	100.00
Magic Sequence 500	192.38	129.80	101.40
Magic Square 6	0.88	133.40	105.20
Partition 32	6 930,00	135.50	101.40
Photo	143.15	131.30	99.60
Queens 100	1.90	132.20	99.30
Crew	3.38	—	191.10
Golf 8-4-9	498.00	—	271.40
Hamming 20-3-32	1 496.00	—	200.70
Steiner 9	124.08	—	191.00

Temporary data structures. One important claim is that iterators are advantageous because they avoid temporary data structures. Table 1 shows in column *temporary* that computing temporary data structures has limited impact (about 3%) on finite domain variables, but considerable impact for set constraints (up to 171% overhead). Temporary data structures have been emulated by wrapping all iterators in a cache-iterator as described in Sect. 5.

Applicability. Deriving several instances from a single propagator implementation significantly reduces the overall amount of code that needs to be written. In Gecode, 31 finite domain propagators are instantiated from 12 generic propagators, 9 Boolean propagators from 4 generic propagators, and 22 set propagators from 9 generic propagators. The generic propagators make up approximately 3800 lines of sources code, saving approximately 4800 lines of code to be written, tested, and maintained.

Obviously, views and iterators are no silver bullet. The mechanism only yields efficient propagators if the compiler can generate the code that would otherwise have been hand-written. If, for example, set complement views are used extensively, the overhead compared to a hand-written propagator can become prohibitive.

10 Conclusion and Future Work

The paper has introduced an architecture decoupling propagators from variables based on views and range iterators. We have argued how to make propagators generic, simpler, and reusable with views for different constraints. We have introduced range iterators as abstractions for efficient domain operations compatible with views. The architecture has been shown to be applicable to many finite domain and finite set constraints. Using parametric polymorphism for views and iterators leads to an efficient implementation that incurs no runtime cost.

Future work. An obvious route for future work is to explore richer variable views. Possible candidates are sums and products of variables going beyond a single variable per view: the challenge here will be to provide efficient range iterators.

This paper explores views only for implementation purposes. A related question is whether views can also be useful for modeling or for automatic transformation of models.

Acknowledgements Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Guido Tack is partially funded by DAAD travel grant D/05/26003. Thanks to Patrick Pekczynski for help with the benchmarks, and to Mikael Lagerkvist for helpful comments. We thank the anonymous reviewers, of this paper and of a previous version, for their constructive comments.

References

1. Pascal Brisset, Hani El Sakkout, Thom Frühwirth, Warwick Harvey, Micha Meier, Stefano Novello, Thierry Le Provost, Joachim Schimpf, and Mark Wallace. ECLiPSe Constraint Library Manual 5.8. User manual, IC Parc, London, UK, February 2005.
2. Gecode: Generic constraint development environment, 2005. Available as an open-source library from `www.gecode.org`.
3. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
4. ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, August 2000.
5. Intelligent Systems Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, April 2005.
6. 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*, number TRA9/00, pages 71–85, 55 Science Drive 2, Singapore 117599, September 2000.
7. Vitaly Lagoon and Peter J. Stuckey. Set domain propagation using ROBDDs. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 347–361, Toronto, Canada, September 2004. Springer-Verlag.
8. Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Fachrichtung Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 2001.
9. 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.
10. Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.
11. Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 513–527, Portland, OR, USA, December 1995. The MIT Press.
12. 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.