Views and Iterators for Generic Constraint Implementations

Christian Schulte[1] and Guido Tack[2]

[1] IMIT, 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. Applications of views exemplify their usefulness and their potential for simplifying constraint implementations. We introduce domain operations compatible with views based on range iterators to access and modify entire variable domains.

1 Introduction

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 propagators (as constraint implementations). Propagators compute generically with variable views instead of variables. Views support operations like scaling, translation, and negation of variables.

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 the construction of propagators by serving as adaptors between variables and propagator datastructures.

The architecture is carefully separated from its implementation. The architecture can be used for arbitrary constraint programming systems and has been fully implemented in Gecode [2].

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.

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 these variables. 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 of other propagators sharing variables.

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.getmin() returns $\min D$, x.getmax() returns $\max D$, x.adjmin(n) updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \geq n\}$, x.adjmax(n) updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \leq n\}$, and x.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 [5], ILOG Solver [7, 8, 3], Eclipse [1], Mozart [6], and Sicstus [4]. Some systems provide additional operations such as for assigning values.

A *domain operation* supports access or update of multiple values of a variable domain simultaneously. In many systems this is provided by supporting an abstract set-datatype for variable domains, as for example in Choco [5], Eclipse [1], Mozart [6], and Sicstus [4]. ILOG Solver [7, 8, 3] only allows access by iterating over the values of a variable domain.

Range notation $[n \mathinner{..} m]$ is used to refer to the set of integers $\{l \in \mathbb{Z} \mid n \leq l \leq m\}$. A *range sequence* $\mathrm{ranges}(I)$ for a finite set of integers $I \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle [n_1 \mathinner{..} m_1], \ldots, [n_k \mathinner{..} m_k] \rangle$ such that I is covered ($\mathrm{set}(s) = I$, where $\mathrm{set}(s)$ is defined as $\bigcup_{i=1}^{k} [n_i \mathinner{..} m_i]$) 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 $\langle [n_i \mathinner{..} m_i] \rangle_{i=1}^{k}$. 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.

Consider as an example the well-known finite domain constraint model for n-Queens using three alldifferent constraints. To be implemented efficiently, this model requires an alldifferent constraint supporting that the values of $x_i + c_i$ are different, where the x_i are variables and the c_i are integers. 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.

For an *offset-view* $v = \mathrm{voffset}(x, c)$ 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:

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

To obtain both alldifferent propagators, also an *identity-view* is needed. An operation on an identity-view $\text{vid}(x)$ 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. 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.

A *scale-view* $v = \text{vscale}(a, x)$ for a positive integer $a > 0$ and a variable x defines operations such that v behaves as $a \cdot x$:

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

As an example, consider the implementation of linear equations. With scale-views it is sufficient to implement the simple propagator $\sum_{i=1}^{n} v_i = c$ for views v_i. Then, a general version $\sum_{i=1}^{n} a_i \cdot x_i = c$ (using scale-views) as well as an optimized version $\sum_{i=1}^{n} x_i = c$ (using identity-views) can be obtained.

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.

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 by $v = \text{vminus}(\text{vid}(x))$.

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 extension is inefficient. 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.

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. Constant-views allow to obtain optimized variants of more general propagators. For example, $x + y = c$ can the be obtained from $x + y + z = c$ without any overhead.

4 Domain Operations and 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 $[n_i \,..\, m_i]$ can be obtained in sequential order but only one at a time. A range iterator r provides the following operations: r.done() tests whether all ranges have been iterated, r.next() moves to the next range, and r.min() and r.max() return the minimum and maximum value for the current range. By set(r) we refer to the set defined by an iterator r (with set(r) = set(s)).

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

Variables are extended with operations to access and modify their domains with range iterators. For a variable x, the operation $x.\,\mathrm{getdom}()$ returns a range iterator for ranges(dom(x)). For a range iterator r the operation $x.\,\mathrm{setdom}(r)$ updates dom(x) to set(r) provided that set(r) $\subseteq$ dom(x).

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

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 all-distinct [9]. After propagation, the new variable domains must be transferred from the data structure to the variables. This can be achieved by using a range iterator as adaptor. The adaptor operates on the data structure and iterates the range sequence for a particular variable. The iterator then can be passed to the appropriate domain operation.

5 Variable Views with Domain Operations

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

Domain operations for an offset-view 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 ioffset(r, c)) are as follows:

$$\begin{array}{ll} o.\mathrm{min}() := r.\mathrm{min}() + c & o.\mathrm{max}() := r.\mathrm{max}() + c \\ o.\mathrm{done}() := r.\mathrm{done}() & o.\mathrm{next}() := r.\mathrm{next}() \end{array}$$

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

$$\begin{array}{l} v.\,\mathrm{getdom}() := \mathrm{ioffset}(x.\,\mathrm{getdom}(), c) \\ v.\,\mathrm{setdom}(r) := x.\,\mathrm{setdom}(\mathrm{ioffset}(r, -c)) \end{array}$$

Providing domain-operations for minus-views and scale-views is similar.

6 Implementation

The presented architecture can be implemented as an orthogonal layer of abstraction for any constraint programming system. The only demand on the implementation language is that it supports *polymorphism* of some kind: propagators operate on different views, domain operations and iterators on different iterators.

C++ features parametric polymorphism through templates. Due to monomorphization, the compiler can perform aggressive optimizations, in particular inlining. Gecode makes heavy use of templates. A thorough inspection of the code generated by several C++ compilers shows that all operations on both views and iterators are inlined entirely. The abstractions thus do not impose a runtime penalty (compared to a system without views and iterators).

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 Mikael Lagerkvist for helpful 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 upon request from the authors, `www.gecode.org`.
3. ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, August 2000.
4. 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.
5. 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.
6. Tobias Müller. *Propagator-based Constraint Solving*. Doctoral dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Fachrichtung Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 2001.
7. 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.
8. 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.
9. 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.