View-based Propagator Derivation (Extended abstract)*

Christian Schulte[1] and Guido Tack[2]

[1] SCALE, KTH Royal Institute of Technology, Sweden
cschulte@kth.se
[2] National ICT Australia (NICTA) and Monash University, Australia
guido.tack@monash.edu

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 effort, but yields better performance. This abstract shows how to use *views* to *derive* propagator variants where derived propagators are *perfect* in that they inherit essential properties such as correctness and domain and bounds consistency. Techniques for systematically deriving propagators are developed, and the abstract sketches an implementation architecture for views that is independent of the underlying constraint programming system. Evaluation of views implemented in Gecode shows that derived propagators are efficient and that views often incur no overhead. Views have proven essential for implementing Gecode, substantially reducing the amount of code that needs to be written and maintained.

1 Introduction

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

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

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

* This is a summary of the paper *View-based propagator derivation* by Christian Schulte and Guido Tack, Constraints 18(1), 75–107 (2013).

Our paper *View-based Propagator Derivation* [9] presents a third approach, introducing *views* to *derive* propagators, which combines the efficiency of dedicated propagator implementations with the simplicity and effortlessness of decomposition. Some preliminary results from this paper were presented as a poster at CP 2008 [8] which contains a formal model of views and derived propagators, which was used to prove that derived propagators inherit properties such as correctness and propagation strength.

This extended abstract focuses on the majority of the material in the journal paper that has not been presented at a conference. In particular, it describes techniques for *systematically* deriving propagators such as transformation, generalization, specialization, and type conversion; it presents an *implementation architecture* for views based on parametric polymorphism; and it summarizes the experimental evaluation that shows that derived propagators are efficient and that views often incur no runtime overhead.

2 Deriving Propagators using Views

We call the basic building block for propagator derivation a *view*. A view can be regarded as a restricted form of a bi-directional *indexical* [2, 10], or an expression such as those supported by IBM ILOG CP Optimizer [4], where the restrictions have been chosen carefully such that the resulting derived propagator satisfies important properties concerning correctness and effectiveness, and such that the implementation does not incur any overhead.

Consider a propagator for the constraint $\max(x,y) = z$. Given three additional propagators for $x' = -x$, $y' = -y$, and $z' = -z$, we could propagate the constraint $\min(x',y') = z'$ using the propagator for $\max(x,y) = z$. Instead of these three additional propagators, we will derive a propagator for max from the propagator for min using views that perform the simple negation transformations.

Views transform input and output of a propagator. For example, a minus view on a variable x transforms the variable domain of x by negating each element. When a propagator *reads* the domain of the minus view, the view returns this transformed domain of x. When the propagator *updates* the domain (e.g. changing a bound or removing a value), the view performs the inverse transformation before updating the domain of x. With views, the implementation of the maximum propagator can be reused: a propagator for the minimum constraint can be derived from a propagator for the maximum constraint and a minus view for each variable.

3 Propagator Derivation Techniques

This section introduces systematic techniques for deriving propagators using views. The techniques capture the transformation, generalization, specialization, and type conversion of propagators. Each technique is illustrated with an example.

Transformation. A transformation view performs a simple operation such as a negation or inversion of a variable domain. The most basic examples are *negation views* for Boolean variables, which correspond to *literals* in SAT solvers. From disjunction $x \vee y = z$ one can then 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.

Scheduling propagators. Many propagation algorithms for constraint-based scheduling [1] are based on tasks t characterized by start time, processing time, and end time. These algorithms are typically expressed in terms of earliest start time ($\text{est}(t)$), latest start time ($\text{lst}(t)$), earliest completion time ($\text{ect}(t)$), and latest completion time ($\text{lct}(t)$).

Another particular aspect of scheduling algorithms is that they are often required in two dual variants. Let us consider not-first/not-last propagation as an example. Assume a set of tasks T and a task $t \notin T$ to be scheduled on the same resource. Then t cannot be scheduled before the tasks in T, if $\text{ect}(t) > \text{lst}(T)$ (where $\text{lst}(T)$ is a conservative estimate of the latest start time of all tasks in T) and hence $\text{est}(t)$ can be adjusted. The dual variant is that t is not-last: if $\text{ect}(T) > \text{lst}(t)$ then $\text{lct}(t)$ can be adjusted. Running the dual variant of a scheduling algorithm on tasks $t \in T$ is the same as running the original algorithm on the *dual tasks* $t' \in T'$, which are simply mirrored at the 0-origin of the time scale: $\text{est}(t') = -\text{lct}(t)$, $\text{ect}(t') = -\text{lst}(t)$, $\text{lst}(t') = -\text{ect}(t)$, and $\text{lct}(t') = -\text{est}(t)$. The dual variant of a scheduling propagator can be automatically derived using a minus view that transforms the time values. In our example, only a propagator for not-first needs to be implemented and the propagator for not-last can be derived (or vice versa).

Generalization. Common views for integer variables capture linear transformations of the integer values: an *offset view* for a variable x and a constant offset o behaves like $x + o$, while a *scale view* for $a \in \mathbb{Z}$ on x behaves like $a \times x$.

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

We can devise an efficient propagation algorithm for the common case of a linear equality constraint with unit coefficients $\sum_{i=1}^{n} x_i = c$. 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.

Specialization. We employ *constant views* to specialize propagators. A constant view behaves like an assigned variable. In practice, specialization has two advantages. Fewer variables require less memory. And specialized propagators can be compiled to more efficient code (constants are known at compile time). Few 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; propagators for the counting constraints $|\{i \mid x_i = c\}| = z$ and $|\{i \mid x_i = y\}| = c$ from a propagator for $|\{i \mid x_i = y\}| = z$; a propagator for set disjointness from a propagator for $x \cap y = z$ and a constant empty set for z.

Type conversion. A type conversion view lets propagators for one type of variable work with a different type, by translating the underlying representation. For example, Boolean variables are essentially integer variables restricted to the values $\{0, 1\}$. CP systems may choose a more efficient implementation for Boolean variables and hence the types for integer and Boolean variables differ. By wrapping an efficient Boolean variable in an *integer view*, all integer propagators can be directly reused.

4 Implementation

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

In an object-oriented implementation of this model, a propagator is an object with a `propagate` method that *accesses* and *modifies* a domain through the methods of variable objects. Such an object-oriented model is used for example by ILOG Solver [7] and Choco [5], and is the basis of most of the current propagation-based constraint solvers.

In order to derive a propagator using view objects, we use *parametricity*, a mechanism provided by the implementation language that supports the instantiation of the same code (the propagator) with different parameters (the views). In C++, for example, a propagator is based on C++ templates, it is *parametric* over the types of the views it uses and can be *instantiated* with any view that provides the necessary operations. This type of parametricity is called *parametric polymorphism*, and is available in other programming languages for example in the form of Java generics [3] or Standard ML functors [6].

In Gecode (version 3.7.2), views are used to derive 616 propagators from 154 propagator implementations. On average, every propagator implementation therefore results in four derived propagators. Propagator implementations in Gecode account for almost 40 000 lines of code and 21 000 lines of documentation. As a rough estimate, deriving propagators using views thus saves around 120 000 lines of code and 60 000 lines documentation to be written, tested, and maintained. Views make up less than 8 000 lines of code, yielding a 1500% return on investment.

5 Experimental Evaluation

This section is a summary of our experimental evaluation of views in Gecode (version 3.7.2).

Our experiments showed that generalization and specialization can be implemented without any performance overhead compared to handwritten propagators. Transformation views on Boolean and integer variables (negation) showed negligible overhead. For set constraints, the basic view operations cannot be fully optimized by the compiler, resulting in an overhead of 32% (geometric mean) across a number of experiments. We also evaluated a decomposition of a constraint compared to a propagator derived using views. The geometric mean of the runtime overhead in this case was 175%, with some examples running almost four times slower. Finally, we evaluated the effect of using parametric polymorphism in C++, comparing it to virtual function calls, which we found to be 28% slower in the geometric mean.

The experiments show that deriving propagators using C++ templates yields competitive (in many cases optimal) performance compared to dedicated handwritten propagators. The results also clarify that deriving propagators is vastly superior to decomposing the constraints into additional variables and simple propagators.

Acknowledgements Christian Schulte has been partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based Scheduling. International Series in Operations Research & Management Science, Kluwer Academic Publishers (2001)
2. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P.H., Kuchen, H. (eds.) Programming Languages: Implementations, Logics, and Programs. LNCS, vol. 1292, pp. 191–206. Springer, Southampton, UK (Sep 1997)
3. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley Professional, third edn. (2005)
4. IBM Corporation: IBM ILOG CP Optimizer V2.3 User's Manual (2009)
5. Laburthe, F.: Choco: Implementing a CP kernel. In: Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T., Perron, L., Schulte, C. (eds.) Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. pp. 71–85 (September 2000)
6. Milner, R., Tofte, M., MacQueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
7. Puget, J.F.: A C++ implementation of CLP. In: Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS). pp. B256–B261. Singapore (Nov 1994)
8. Schulte, C., Tack, G.: Perfect derived propagators. In: Stuckey, P.J. (ed.) Fourteenth International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 5202, pp. 571–575. Springer, Sydney, Australia (Sep 2008)
9. Schulte, C., Tack, G.: View-based propagator derivation. Constraints 18(1), 75–107 (2013), http://dx.doi.org/10.1007/s10601-012-9133-z
10. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). Journal of Logic Programming 37(1-3), 293–316 (1998)