View-based Propagator Derivation

Christian Schulte

SCALE, KTH & SICS, Sweden

joint work with:

Guido Tack

NICTA & Monash University, Australia

Based on: **View-based Propagator Derivation.** Christian Schulte, Guido Tack. *Constraints* 18(1), pages 75-107. Springer-Verlag, January, 2013. DOI 10.1007/s10601-012-9133-z.

Building a CP System

advanced search heuristics

no-goods

interfaces

many constraints

many examples

graphical tools

many variable types

parallel search

complete reference documentation

extensive tutorial documentation

restart-based search

fast, simple, and generic kernel

comprehensive tests

How Many Tries Do You Have?

1

- Have to get it right first time potential user downloads
- Decisions for Gecode 1.0.0:
    - sufficiently **many** constraints
    - sufficiently **high** speed
    - sufficiently **few** bugs
    - release as open source **on time**
    - accessible **to experts**

Gecode 1.0.0

advanced search heuristics

no-goods

interfaces

many constraints

many examples

graphical tools

many variable types

parallel search

complete reference documentation

extensive tutorial documentation

restart-based search

fast, simple, and generic kernel

comprehensive tests

This Talk

advanced search heuristics

no-goods

interfaces

many constraints

many examples

graphical tools

many variable types

parallel search

complete reference documentation

extensive tutorial documentation

restart-based search

fast, simple, and generic kernel

comprehensive tests

Which Constraints?

- A propagator for

  $\min(x_1, \ldots, x_n)=y$

  as well as

  $\max(x_1, \ldots, x_n)=y$?
- A propagator for

  $a_1 \times x_1 + \ldots + a_n \times x_n = c$ (a_i, c integers)

  as well as

  $x_1 + \ldots + x_n = c$?
- A propagator for

  $(x_1 + \ldots + x_n = c) \leftrightarrow y$ (c integer)

  as well as

  $(x_1 + \ldots + x_n \neq c) \leftrightarrow y$? (c integer)

Decompose Constraints? No!

- Decompose

  $\max(x_1, \ldots, x_n)=y$

  into

  $\min(z_1, \ldots, z_n)=u \wedge x_1= -z_1 \wedge \ldots \wedge x_n= -z_n \wedge y= -u$

  - **no way: clashes with speed and fast kernel**

- Decompose

  $a_1 \times x_1 + \ldots + a_n \times x_n = c$ (a_i, c integers)

  into

  $y_1 + \ldots + y_n=c \wedge y_1= a \times x_1 \wedge \ldots \wedge y_n= a \times x_n$

  - **absolutely no way: yields less propagation**

Implement Propagators? No!

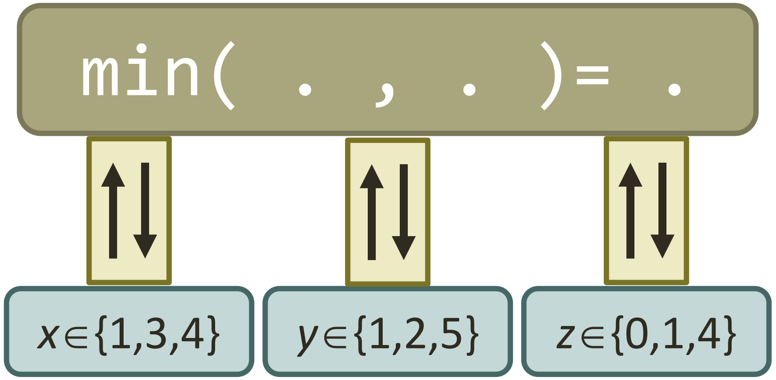
- Tremendous effort to implement propagator variants
  - Gecode: just three people
  - research interest is **not** implementing constraints
- Additional effort for
  - documentation
  - testing
  - maintenance
- Effort potentially prohibitive

Derive Propagators? Yes!

- **Derive** propagators using **variable views**
- Using **systematic derivation** techniques
- View idea
  - folded into propagator
  - bi-directional mapping of values
- Derived propagators are **perfect**
  - correctness
  - propagation strength: bounds and domain consistency
  - implementation aspects: fixpoints and subsumption
  - little overhead (often none)

HOW AND WHY VIEWS WORK

Propagating max with Views



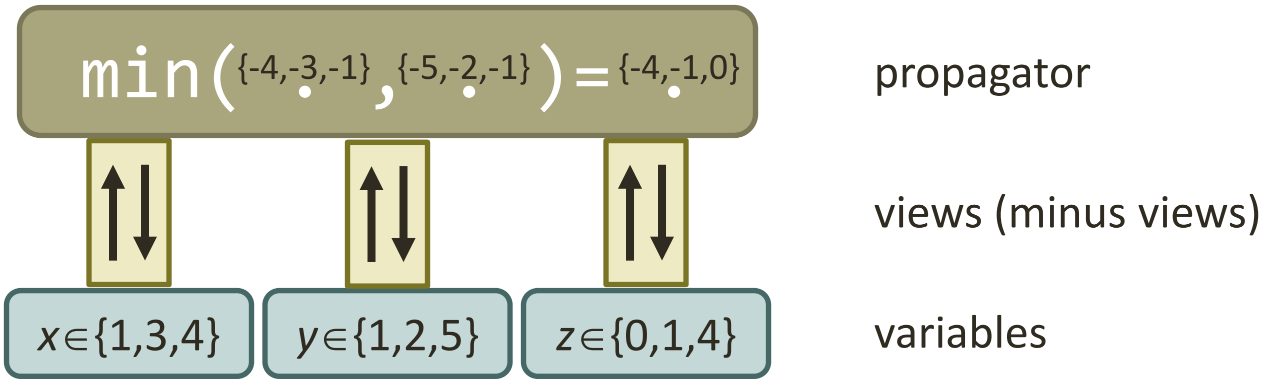
propagator

views (minus views)

variables

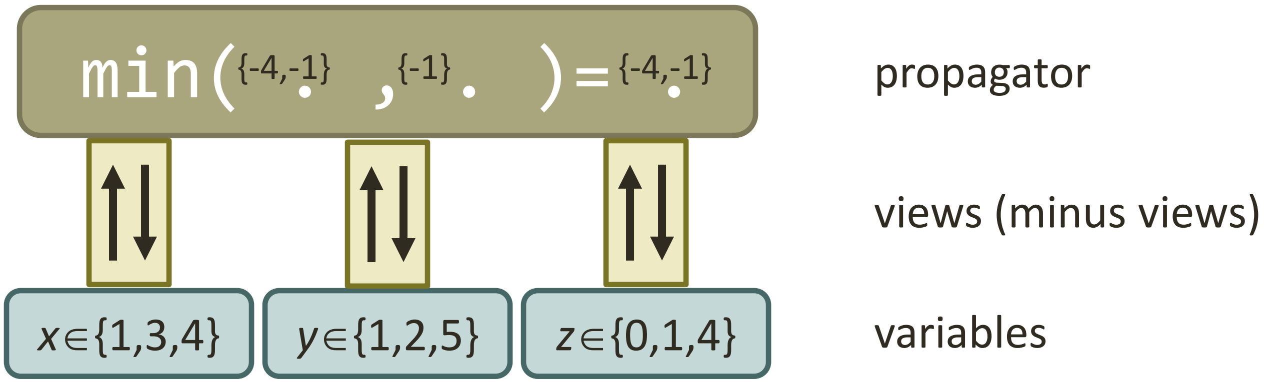
- Propagator

Propagating max with Views



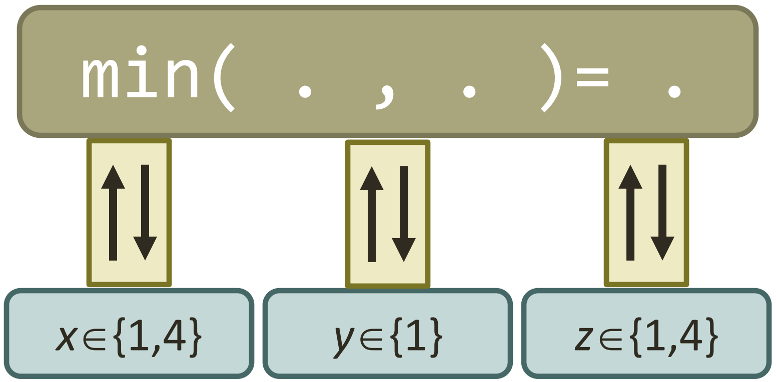
- Propagator
  1. reads values through **views**

Propagating max with Views



- Propagator
  1. reads values through **views**
  2. performs propagation wrt values read

Propagating max with Views



propagator

views (minus views)

variables

- Propagator
  1. reads values through **views**
  2. performs propagation wrt values read
  3. writes values through **inverse** of **views**

Model for Views

- **Variable view** for a variable x

  $$\varphi_x : V \rightarrow V'$$

  injective function from values V to values V'
  - different value sets matter

- **View** φ is a family of variable views φ_x for all variables x

- Possible to define **inverse view** φ^-

- Propagator **derived** from propagator p

  $$\varphi^- \bullet p \bullet \varphi$$

- Also define **derived constraint**, …

Facts

- Derived propagator
  - is in fact a propagator (preserves contraction and monotonicity)
  - implements the “right” constraint (constraint composed with views)
  - preserves fixpoints and subsumption
  - inherits domain-consistency
- With additional (natural) requirements inherits
  - bounds(**Z**)-consistency
  - bounds(D)-consistency
  - depends on whether hull operator commutes with view

Limitations

- Views are injective
  - generalization might make propagators non contracting
  - studied in [1,2]
- Views map values for single variable
  - generalization might make propagators non contracting
  - studied in [1]
- Propagator invariants might be violated (rare)
  - propagators typically rely on variable domain invariants
  - example
    - adjusting lower bound of set variable does not change upper bound
    - might be violated by view

[1] Correia, Barahona. View-based propagation of decomposable constraints, Constraints, 2013.
[2] Van Hentenryck, Michel. Domain views for constraint programming, CP 2014.

Not a Limitation

- Approach works **for any propagator**

- No restriction to bounds consistency

Techniques: transformation, generalization, specialization, type conversion, enforcing invariants

USING VIEWS

Transformation: Boolean

- Use **negation views** defined as

  $\varphi_x(v) = 1 - v$

  to derive
  - $\neg x = y$ from $x = y$ (on x)
  - $x \wedge y = z$ from $x \vee y = z$ (on x, y, z)
  - $x \rightarrow y = z$ from $x \vee y = z$ (on x)
  - $x \oplus y = z$ from $x \leftrightarrow y = z$ (on z)

  - $x_1 \wedge \ldots \wedge x_n \wedge \neg y_1 \wedge \ldots \wedge \neg y_n = z$

    from $x_1 \vee \ldots \vee x_n \vee \neg y_1 \vee \ldots \vee \neg y_n = z$ (on all)

  - $x_1 \wedge \ldots \wedge x_n \wedge \neg y_1 \wedge \ldots \wedge \neg y_n = 0$

    from $x_1 \vee \ldots \vee x_n \vee \neg y_1 \vee \ldots \vee \neg y_n = 1$ (on all)
    - optimized implementation with watched literals re-used!

Transformation: Boolean

- Use negation views to derive
  - $x_1 + \ldots + x_n \leq c$ from $x_1 + \ldots + x_n \geq c$ (c integer)

    as $x_1 + \ldots + x_n \leq c \Leftrightarrow \neg x_1 + \ldots + \neg x_n \geq n - c$
  - $(x_1 + \ldots + x_n \neq c) \leftrightarrow y$ (c integer)

    from $(x_1 + \ldots + x_n = c) \leftrightarrow y$
    - same idea for many reified constraints

Transformation: Set

- **Complement view**: analogous to negation view
- Intersection from union, set difference from union, …

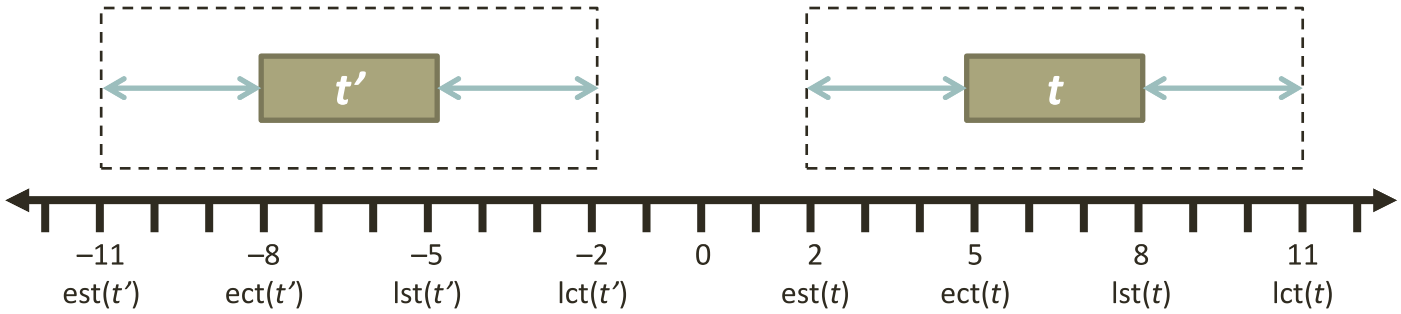
Transformation: Integer

- Use **minus views** defined as

  $\varphi_x(v) = -v$

  to derive
  - $\min(x, y) = z$ from $\max(x, y) = z$ (on x, y, z)
    - bounds-consistent propagator (bounds(**Z**))
    - domain-consistent propagator
  - $\min(x_1, \ldots, x_n) = y$ from $\max(x_1, \ldots, x_n) = y$ (on $x_1, \ldots, x_n, y$)
    - bounds-consistent propagator (bounds(**Z**))
    - domain-consistent propagator

Transformation: Scheduling



- Scheduling propagators implemented in terms of

  est(t) ≡ earliest start time ect(t) ≡ earliest completion time
  lst(t) ≡ latest start time lct(t) ≡ latest completion time
- Two variants needed
  - primary: t is not first → adjust est(t)
  - dual: t is not last → adjust lct(t)
- Dual can be derived with minus views (mirror at 0-origin)

  est(t') = –lct(t), ect(t') = –lst(t), lst(t') = –ect(t), lct(t') = –est(t)
- Can reuse complex data structures, for example Ω-trees

  [Vilím. O(n log n) filtering algorithms for unary resource constraints, CP AI OR 2004]

Generalization

- Use **scale view** for integer *a* defined as

  $\varphi_x(v) = a \times v$

  to derive

  $a_1 \times x_1 + \ldots + a_n \times x_n = c$

  from

  $x_1 + \ldots + x_n = c$

- Use **offset view** for integer *o* defined as

  $\varphi_x(v) = v + o$

  to derive

  alldifferent($x_1 + c_1, \ldots, x_n + c_n$)

  from

  alldifferent($x_1, \ldots, x_n$)

Specialization

- **Constant view** behaves like an assigned variable
  - less memory
  - more efficient code if constant known at compile time

- Derive
  - $x + y \leq c$ from $x + y + z \leq c$ (use 0 for z)
  - $(x = c) \leftrightarrow b$ from $(x = y) \leftrightarrow b$ (use c for y)
  - $|\{ i \mid x_i = c\}| = z$ and $|\{ i \mid x_i = y\}| = c$ from $|\{ i \mid x_i = y\}| = z$ (use c for y or z)
  - disjoint(x,y) from $x \cap y = z$ (use $\emptyset$ for z)

Type Conversion

- **Integer view** wraps Boolean 0/1 variable as an integer variable
  - 0/1 variables might have a more efficient implementation
  - all integer propagators can now be on 0/1 variables
  - some propagators should be still specific to 0/1 variables (linear inequalities due to watched literals, …)
- **Singleton view** wraps integer variable as a set variable
  - derive $x \in y$ from $x \subseteq y$

Enforcing Invariants

- Assume bounds(**Z**)-consistent propagator for $x \times y = z$
  - propagation depends on whether 0 in x, 0 in y, or 0 in z
- Direct implementation: convoluted and inefficient
- Rewriting: replace propagator if 0 excluded
  - $x > 0 \wedge y > 0 \wedge z > 0$, or
  - $x > 0 \vee y > 0$, or
  - $z > 0$

  requires three different propagators
- Derivation: derive all from single propagator with minus views

Enforcing Invariants

- Implementation of

  binpacking($l_1, \ldots, l_m, b_1, \ldots, b_n, s_1, \ldots, s_n$)

  with
  - load variables: l_j is the load of bin j
  - bin variables: item i with size s_i is packed into bin b_i

    [Shaw. A constraint for bin packing, CP 2004]

- Enforce that all items are not yet packed
  - use offset views $l_j + c_j$ for load variables
  - when item i is packed into bin j (bin variable b_i assigned to j):

    subtract size s_i from load offset c_j

    eliminate item i from b and s
  - simplifies implementation and saves memory

IMPLEMENTATION IDEA

Integer Variable

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

- Object-oriented model (ILOG Solver, Choco, Gecode, …)
  - variables are objects
  - propagators are objects

    [Puget. A C++ Implementation of CLP, SPICIS 1994]

Minus View

```
class MinusView {
protected: IntVar* x;
public:    MinusView(IntVar* x0)
             : x(x0) {}
           int min(void) { return -x->max(); }
           int max(void) { return -x->min(); }
           void adjmin(int n) {
             x->adjmax(-n);
           }
           void adjmax(int n) { … }
};
```

- Implements exactly same interface

Propagator

```
template<class VX, class VY>
class LessThan : public Propagator {
protected: VX* x; VY* y;
public:    LessThan(VX* x0, VY* y0) : x(x0), y(y0) { … }
           virtual void propagate(void) {
             x->adjmax(y->max()-1);
             y->adjmin(x->min()+1);
           }
};
```

- Propagators are parametric with respect to their views
- Creation of `x < y`:

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

- Creation of `x > y`:

```
new LessThan<MinusView,MinusView>(new MinusView(x),
                                  new MinusView(y));
```

Choice of Parametricity

- Two variants available
  - parametric polymorphism (templates in C++, higher-order functions in Haskell, …) **more efficient**
  - dynamic binding (virtual functions in C++, methods in Java) **more expressive**
- Gecode uses templates in C++
  - polymorphism resolved at compile time
  - some views optimized away entirely

Domain Operations

- Operations that adjust the whole domain
- Efficient architecture based on range iterators
  - each range (interval) can be obtained in sequential order, one at a time
  - no data structures required
  - operations for views easily defined per range
- Details and evaluation in paper

COST AND BENEFITS

Return on Investment

variable type	implemented	derived	ratio
integer	93	377	4.05
Boolean 0/1	30	93	3.10
integer set	31	146	4.71
overall	**154**	**616**	**4.00**

- Propagator implementations: ≈ 40 000 lines of code
  ≈ 21 000 lines of documentation
- Views save : ≈ 120 000 lines of code
  ≈ 60 000 lines of documentation
- View implementations: ≈ 8 000 lines of code and doc
- Return on investment: ≈ 1 500 %

[Gecode 3.7.2]

Evaluation Summary

- Decomposition always worse than views

integer benchmarks	126% more time	(14% … 485%)
	101% more memory	(2% … 267%)
set benchmarks	46% more time	(12% … 131%)
	31% more memory	(2% … 144%)

- Minus and negation views for free
  - often optimized away by compiler
  - benchmarks using minus views on alldifferent confirm
- Complement view for sets not for free
  - 32% overhead compared to handwritten propagators

Evaluation Summary

- Virtual methods worse than templates
  - integer benchmarks 34% more time (5% … 118%)
  - set benchmarks 18% more time (9% … 126%)

  - template = compile-time polymorphism
  - virtual method = run-time polymorphism

SUMMARY

Related Approaches

- Indexicals and ILOG expressions
  - indexicals: uni-directional, more expressive
  - expressions: bi-directional, more expressive, no guarantees on update
- SAT literals use negation views (for example MiniSat)
- Views in other systems
  - Minion, CaSPER, Objective CP
  - useful for lazy clause generation

[References and extensive discussion in paper]

Take Home

- Views = useful compromise

  efficiency ⇆ expressiveness

- Systematic derivation techniques

- Can be build on top of any system
  - needs some form of parametricity

- Gecode without views would have been…

  slow or **impossible**

Integer Variable

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

- Object-oriented model (ILOG Solver, Choco, Gecode, …)
  - variables are objects
  - propagators are objects

    [Puget. A C++ Implementation of CLP, SPICIS 1994]

Offset View

```
class OffsetView {
protected: IntVar* x; int c;
public:    OffsetView(IntVar* x0, int c0)
             : x(x0), c(c0) {}
           int min(void) { return x->min()+c; }
           int max(void) { return x->max()+c; }
           void adjmin(int n) {
             x->adjmin(n-c);
           }
           void adjmax(int n) { … }
           void subscribe(EventSet e) {
              x->subscribe(e);
           }
};
```

- Implements same interface as variable

Minus View

```
class MinusView {
protected: IntVar* x;
public:    MinusView(IntVar* x0)
             : x(x0) {}
           int min(void) { return -x->max(); }
           int max(void) { return -x->min(); }
           void adjmin(int n) {
             x->adjmax(-n);
           }
           void adjmax(int n) { … }
           void subscribe(EventSet e) {
              x->subscribe(negate(e));
           }
};
```

Constant View

```
class ConstView {
protected: int c;
public:    ConstView(int c0)
             : c(c0) {}
           int min(void) { return c; }
           int max(void) { return c; }
           void adjmin(int n) {
             if (n > c) fail();
           }
           void adjmax(int n) { … }
           void subscribe(EventSet e) {
             schedule();
           }
};
```

Propagator

```
template<class VX, class VY>
class LessThan : public Propagator {
protected: VX* x; VY* y;
public:    LessThan(VX* x0, VY* y0) : x(x0), y(y0) {
             x->subscribe(LOWER_BOUND); y->subscribe(UPPER_BOUND);
           }
           virtual void propagate(void) {
             x->adjmax(y->max()-1); y->adjmin(x->min()+1);
           }
};
```

- Propagators are parametric with respect to their views
- Creation of `x < y`:

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

- Creation of `x > y`:

```
new LessThan<MinusView,MinusView>(new MinusView(x),
                                  new MinusView(y));
```