6 Modeling convenience: MiniModel

MiniModel (see Direct modeling support) provides some little helpers to the constraint modeler. However, it does not offer any new constraints or branchers.

Important. Do not forget to add

```
#include <gecode/minimodel.hh>
```

to your program when you want to use MiniModel. Note that the same conventions hold as in Chapter 4.

6.1 Linear expressions and relations

Linear expressions and relations are constructed according to the structure sketched in Figure 6.1, where the standard C++ operators are used (for an example, see Section 3.1). Linear expressions and relations can be constructed over both integer and Boolean variables.

The only purpose of linear expressions and relations is posting constraints defined by them (see Posting of expressions and relations). Given a linear *expression*, the `expr` function returns a new integer variable (regardless of whether the expression is over integer or Boolean variables) constrained to the value of the expression. The initial domain of the new variable is chosen to be just large enough to fit the value of the expression. The constraint posted for the expression will be as few `linear` constraints as possible (see Section 4.4.5) to ensure maximal constraint propagation.[1]

For example, the linear expression `a*x-4*y+2` where both `x` and `y` are integer variables and `a` is an integer is posted by

```
IntVar z=expr(home, a*x-4*y+2);
```

An important aspect of posting an expression is that the returned variable is initialized with a reasonably small variable domain, see Tip 4.3.

A linear *relation* can be posted using the `rel` function, which just posts the corresponding `linear` constraint. Assume that `z` is also an integer variable, then

[1]In case the expression has only integer variables or only Boolean variables, a single `linear` constraint is posted. If the expression contains both integer and Boolean variables, two `linear` constraints are posted.

$\langle \mathit{LinExpr} \rangle$	::=	$\langle n \rangle$	integer value
	|	$\langle x \rangle$	integer or Boolean variable
	|	`-`$\langle \mathit{LinExpr} \rangle$	unary minus
	|	$\langle \mathit{LinExpr} \rangle$ `+` $\langle \mathit{LinExpr} \rangle$	addition
	|	$\langle \mathit{LinExpr} \rangle$ `-` $\langle \mathit{LinExpr} \rangle$	subtraction
	|	$\langle n \rangle * \langle \mathit{LinExpr} \rangle$	linear multiplication
	|	$\langle \mathit{LinExpr} \rangle * \langle n \rangle$	linear multiplication
	|	`sum(`$\langle \overline{x} \rangle$`)`	sum of integer or Boolean variables
	|	`sum(`$\langle \overline{n} \rangle$`,`$\langle \overline{x} \rangle$`)`	sum of integer or Boolean variables with integer coefficients
$\langle \mathit{LinRel} \rangle$	::=	$\langle \mathit{LinExpr} \rangle$ $\langle r \rangle$ $\langle \mathit{LinExpr} \rangle$	linear relation
$\langle r \rangle$	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`	relation symbol
$\langle \overline{x} \rangle$	::=	array of integer or Boolean variables	
$\langle \overline{n} \rangle$	::=	array of integers	

Figure 6.1: Linear expressions and relations

```
rel(home, z == a*x-4*y+2);
```

posts the same constraint as in the previous example.

Even arrays of variables (possibly with integer argument arrays as coefficients) can be used for posting linear expressions and relations. For example, if x and y are integer variables and z is an array of integer variables, then

```
rel(home, x+2*sum(z) < 4*y);
```

posts a single linear constraint that involves all variables from the array z.

Like the post functions for integer and Boolean variables presented in Section 4.4, posting linear expressions and relations supports an optional argument of type `IntConLevel` to select the consistency level. For more information, see Section 4.3.

For examples, see Alpha puzzle, SEND+MORE=MONEY puzzle, Chapter 30, Chapter 32, and Section 3.1.

6.2 Boolean expressions and relations

Boolean expressions and relations are constructed using standard C++ operators according to the structure sketched in Figure 6.2.

Again, the only purpose of a Boolean expression or relation is to post a corresponding constraint for it (see Posting of expressions and relations). Posting a Boolean expression returns a new Boolean variable that is constrained to the value of the expression. Several

⟨*BoolExpr*⟩	::=	⟨*x*⟩	Boolean variable
	|	! ⟨*BoolExpr*⟩	negation
	|	⟨*BoolExpr*⟩ && ⟨*BoolExpr*⟩	conjunction
	|	⟨*BoolExpr*⟩ || ⟨*BoolExpr*⟩	disjunction
	|	⟨*BoolExpr*⟩ == ⟨*BoolExpr*⟩	equivalence
	|	⟨*BoolExpr*⟩ != ⟨*BoolExpr*⟩	non-equivalence
	|	⟨*BoolExpr*⟩ >> ⟨*BoolExpr*⟩	implication
	|	⟨*BoolExpr*⟩ << ⟨*BoolExpr*⟩	reverse implication
	|	⟨*BoolExpr*⟩ ˆ ⟨*BoolExpr*⟩	exclusive or
	|	⟨*LinRel*⟩	reified linear relation
	|	⟨*SetRel*⟩	reified set relation

Figure 6.2: Boolean expressions

constraints might be posted for a single expression, however as few constraints as possible are posted. For example, all negation constraints are eliminated by rewriting the Boolean expression into NNF (negation normal form) and conjunction and disjunction constraints are combined whenever possible.

For example, the Boolean expression `x && (y >> z)` (to be read as $x \land (y \to z)$) for Boolean variables x, y, and z is posted by

```
BoolVar b=expr(home, x && (y >> z));
```

Tip 6.1 (Boolean precedences)**.** Note that the precedences of the Boolean connectives are different from the usual mathematical notation. In C++, operator precedence cannot be changed, so the precedences are as follows (high to low): `!`, <<, >>, ==, `!=`, ˆ, &&, `||`. For instance, this means that the expression $b_0 == b_1 >> b_2$ will be interpreted as $(b_0 \leftrightarrow b_1) \to b_2$ instead of the more canonical $b_0 \leftrightarrow (b_1 \to b_2)$. If in doubt, use parentheses! ◄

Any Boolean expression e corresponds to the Boolean relation stating that e is true. Posting a Boolean relation posts the corresponding Boolean constraint. Using the Boolean expression from above,

```
rel(home, x && (y >> z));
```

posts that $x \land (y \to z)$ must be true, whereas

```
rel(home, !(x && (y >> z)));
```

posts that $x \land (y \to z)$ must be false.

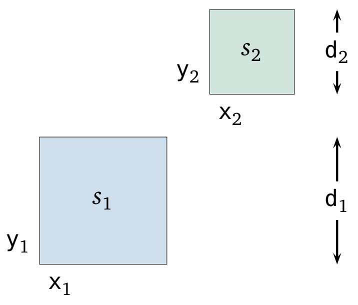
Boolean expressions include reified linear relations. As an example consider the placement of two squares s_1 and s_2 such that the squares do not overlap. A well known model for this constraint is

$$\begin{array}{ccccc} x_1 + d_1 \le x_2 & \lor & x_2 + d_2 \le x_1 & \lor \\ y_1 + d_1 \le y_2 & \lor & y_2 + d_2 \le y_1 & \end{array}$$

C++ expression	function	bounds	domain
`min(x, y)`	$\min(x, y)$	✓	✓
`max(x, y)`	$\max(x, y)$	✓	✓
`abs(x)`	$\lvert x \rvert$	✓	✓
`x * y`	$x \cdot y$	✓	✓
`sqr(x)`	x^2	✓	✓
`sqrt(x)`	$\lfloor \sqrt{x} \rfloor$	✓	✓
`x / y`	$x \div y$	✓	
`x % y`	$x \bmod y$	✓	
`element(x,y)`	$x[y]$	✓	✓

Figure 6.3: Non-linear arithmetic functions (x and y are linear expressions)

The meaning of the integer variables x_i and y_i, and the integer values d_i is sketched to the right. The squares do not overlap, if the relative position of s_1 with respect to s_2 is either left, right, above, or below. As soon as one of the relationships is established, the squares do not overlap.
With Boolean relations using reified linear relations, the constraint that the squares s_1 and s_2 do not overlap can be posted as follows:



```
rel(home, (x1+d1 <= x2) || (x2+d2 <= x1) ||
          (y1+d1 <= y2) || (y2+d2 <= y1));
```

Like the post functions for integer and Boolean variables presented in Section 4.4, posting Boolean expressions and relations supports an optional argument of type `IntConLevel` to select the consistency level. For more information, see Section 4.3.

For more examples using Boolean expressions and relations including reification, see Chapter 34, Packing squares into a rectangle, and The balanced academic curriculum problem.

Boolean expressions also include set relations, which will be covered in Section 6.7.

6.3 Non-linear arithmetic functions

Arithmetic functions provide functions and operators for non-linear expressions such as multiplication of two variables, minimum or maximum. The available functions and operators are listed in Figure 6.3.

These non-linear expressions are translated into linear expressions by *decomposition*. For example, posting the constraint

```
rel(home, a+b*(c+d) == 0);
```

alias	constraint posted
`atmost(home, x, u, v);`	`count(home, x, u, IRT_LQ, v);`
`atleast(home, x, u, v);`	`count(home, x, u, IRT_GQ, v);`
`exactly(home, x, u, v);`	`count(home, x, u, IRT_EQ, v);`
`lex(home, x, r, y);`	`rel(home, x, r, y);`

Figure 6.4: Aliases for integer constraints (x and y are integer variable arrays, u and v are integers or integer variables, r is an integer relation type)

for integer variables a, b, c, and d is equivalent to the decomposition

```
IntVar tmp0 = expr(home, c+d);
IntVar tmp1 = expr(home, b*tmp0);
rel(home, x+tmp1 == 0);
```

The last two columns in Figure 6.3 specify which consistency levels are supported by the decomposed expressions. For example,

```
rel(home, x+expr(home,y*z,ICL_DOM) == 0);
```

will propagate domain consistency for the multiplication, but bounds consistency for the sum.

6.4 Channeling functions

Channel functions are functions to channel a Boolean variable to an integer variable and vice versa. For an integer variable x,

```
channel(home, x);
```

returns a new Boolean variable that is equal to x. Likewise, for a Boolean variable x an equal integer variable is returned.

6.5 Aliases for integer constraints

Aliases for integer constraints provide some popular aliases. Figure 6.4 lists the aliases and their corresponding definitions.

6.6 Regular expressions for extensional constraints

Regular expressions are implemented as instances of the class REG and provide an alternative, typically more convenient, interface for the specification of extensional constraints than DFAs do. The construction of regular expressions is summarized in Figure 6.5.

operation	meaning
`REG r`	initialize `r` as ϵ (empty)
`REG r(4)`	initialize `r` as single integer (symbol) `4`
`REG r(IntArgs(3, 0,2,4))`	initialize `r` as alternative of integers `0`|`2`|`4`
`r + s`	`r` followed by `s`
`r | s`	`r` or `s`
`r += s`	efficient shortcut for `r = r + s`
`r |= s`	efficient shortcut for `r = r | s`
`*r`	repeat `r` arbitrarily often (Kleene star)
`+r`	repeat `r` at least once
`r(n)`	repeat `r` at least `n` times
`r(n,m)`	repeat `r` at least `n` times, at most `m` times

Figure 6.5: Constructing regular expressions (`r` and `s` are regular expressions, `n` and `m` are integers)

Let us reconsider the Swedish drinking protocol from Section 4.4.11. The protocol can be described by a regular expression `r` constructed by

```
REG r = *REG(0) + *(REG(1) + +REG(0));
```

A sequence of activities `x` (an integer or Boolean variable array) can be constrained by

```
DFA d(r);
extensional(home, x, d);
```

after a DFA for the regular expression has been computed.

Tip 6.2 (Creating a DFA only once)**.** Please make it a habit to create a DFA explicitly from a regular expression `r` rather than implicitly by

```
extensional(home, x, r);
```

Both variants work, however the implicit variant disguises the fact that each time the code fragment is executed, a new DFA for the regular expression `r` is computed (think about the code fragment being executed inside a loop and your C++ compiler being not too smart about it)![2] ◀

For examples on using regular expressions for extensional constraints, see the nonogram case study in Chapter 33 or the examples Solitaire domino, Nonogram, and Pentominoes. The models are based on ideas described in [13], where regular expressions for extensional constraints nicely demonstrate their usefulness.

[2]The integer module cannot know anything about regular expressions. Hence, it is impossible in C++ to avoid the implicit conversion. This is due to the fact that the conversion is controlled by a type operator (that must reside in the MiniModel module) and not by a constructor that could be made **`explicit`**.

⟨*SetExpr*⟩	::=	⟨*x*⟩	set variable
	|	- ⟨*SetExpr*⟩	complement
	|	⟨*SetExpr*⟩ & ⟨*SetExpr*⟩	intersection
	|	⟨*SetExpr*⟩ | ⟨*SetExpr*⟩	union
	|	⟨*SetExpr*⟩ + ⟨*SetExpr*⟩	disjoint union
	|	⟨*SetExpr*⟩ - ⟨*SetExpr*⟩	set difference
⟨*SetRel*⟩	::=	⟨*SetExpr*⟩ == ⟨*SetExpr*⟩	expressions are equal
	|	⟨*SetExpr*⟩ <= ⟨*SetExpr*⟩	first is subset of second expression
	|	⟨*SetExpr*⟩ >= ⟨*SetExpr*⟩	first is superset of second expression
	|	⟨*SetExpr*⟩ || ⟨*SetExpr*⟩	expressions are disjoint

Figure 6.6: Set expressions and relations

6.7 Set expressions and relations

Set expressions and relations are constructed using standard C++ operators as sketched in Figure 6.6. Just like for linear and Boolean expressions, posting of a set expression returns a new set variable that is constrained to the value of the expression. For example, the set expression `x & (y|z)` (to be read as $x \cap (y \cup z)$) for set variables `x`, `y`, and `z` is posted by

```
SetVar s = expr(home, x & (y | z));
```

Posting a set relation posts the corresponding constraint. Given an existing set variable `s`, the previous code fragment could therefore be written as

```
rel(home, s == (x & (y | z)));
```

Set relations can be reified, turning them into Boolen expressions. The following code posts the constraint that `b` is true if and only if `x` is the complement of `y`:

```
BoolVar b = expr(home, (x == -y));
```

Intead of a set variable, you can always use a constant `IntSet`, for example for reifying the fact that `x` is empty:

```
BoolVar b = expr(home, (x == IntSet::empty));
```

The subset relations can also be posted two-sided, such as

```
rel(home, IntSet(0,10) <= x <= IntSet(0,20));
```

MiniModel provides three convenience functions for posting constraints that link set variables and integer variables. Assume that `x` is an integer variable and `y` is a set variable. Then, `x` is constrained to be less than the smallest element of `y` by

```
rel(home, x < min(home,y));
```

To constrain x to be different from the largest element of y, the following relation can be posted:

```
rel(home, x != max(home,y));
```

The following enforces that the cardinality of y equals x:

```
rel(home, x == cardinality(home,y));
```

6.8 Matrix interface for arrays

MiniModel provides a Matrix support class for accessing an array as a two dimensional matrix. The following

```
IntVarArgs x(n*m);
...
Matrix<IntVarArgs> mat(x, n, m);
```

declares an array of integer variables x and superimposes a matrix interface to x called mat with width n and height m. Note that the first argument specifies the number of columns, and the second argument specifies the number of rows.

The elements of the array can now be accessed at positions $\langle i, j \rangle$ in the matrix mat (that is, the element in column i and row j) using

```
IntVar mij = mat(i,j);
```

Furthermore, the rows and columns of the matrix can be accessed using mat.row(i) and mat.col(j). If a rectangular slice is required, the slice() member function can be used.

A matrix interface can be declared for any standard array or argument array used in Gecode, such as IntVarArray or IntSetArgs.

As an example of how the Matrix class can be used, consider the Sudoku problem (see Solving Sudoku puzzles using both set and integer). Given that there is a member IntVarArray x that contains $9 \cdot 9$ integer variables with domain $\{1, \ldots, 9\}$, the following code posts constraints that implement the basic rules for a Sudoku.

```
Matrix<IntVarArray> m(x, 9, 9);

for (int i=0; i<9; i++)
  distinct(home, m.row(i));
for (int i=0; i<9; i++)
  distinct(home, m.col(i));
for (int i=0; i<9; i+=3)
  for (int j=0; j<9; j+=3)
    distinct(home, m.slice(i, i+3, j, j+3));
```

For more examples that use the Matrix class, see Chapter 38, Chapter 36, Chapter 37, Chapter 33, Magic squares, and Nonogram.

Element constraints. A matrix can also be used with an element constraint that propagates information about the row and column of matrix entries.

For example, the following code assumes that x is an integer array of type IntArgs with 12 elements.

```
Matrix<IntArgs> m(x, 3, 4);
IntVar r(home,0,1024), c(home,0,1024), v(home,0,1024);
element(home, m, r, c, v);
```

constrains the variable v to the value at position $\langle \mathtt{r}, \mathtt{c} \rangle$ of the matrix m.

Tip 6.3 (Element for matrix can compromise propagation)**.** Whenever it is possible one should use an array rather than a matrix for posting element constraints, as an element constraint for a matrix will provide rather weak propagation for the row and column variables.

Consider the following array of integers x together with its matrix interface m

```
IntArgs x(4, 0,2,2,1);
Matrix<IntArgs> m(x,2,2);
```

That is, m represents the matrix

$$\begin{pmatrix} 0 & 2 \\ 2 & 1 \end{pmatrix}$$

Consider the following example using an element constraint on an integer array:

```
IntVar i(home,0,8), v(home,0,1);
element(home, x, i, v);
```

After performing propagation, i will be constrained to the set $\{0, 3\}$ (as 2 is not included in the values of v).

Compare this to propagating an element constraint over the corresponding matrix as follows:

```
IntVar r(home,0,8), c(home,0,8), v(home,0,1);
element(home, m, r, c, v);
```

Propagation of element will determine that only the fields $\langle 0, 0 \rangle$ and $\langle 1, 1 \rangle$ are still possible. But propagating this information to the row and column variables, yields the values $\{0, 1\}$ for both r and c: each value for the coordinates is still possible even though some of their combinations are not. ◀

6.9 Support for cost-based optimization

Support for cost-based optimization provides two subclasses of Space, MinimizeSpace and MaximizeSpace to search for a solution of minimal and maximal, respectively, cost. In order to use these abstract classes, a class inheriting from them must implement a virtual cost function of type

```
virtual IntVar cost(void) const { ··· }
```

The function must return an integer variable for the cost. For an example, see Section 3.2.

Tip 6.4 (Cost must be assigned for solutions)**.** In case the `cost()` function is called on a *solution*, the variable returned by `cost()` *must* be assigned. If the variable is unassigned for a solution, an exception of type `Int::ValOfUnassignedVar` is thrown. ◄