



API

1. Introduction

This document describes the interface of the SUGAR fruit of the OCRE project. It offers useful simple constraints that are not implemented in the CHOCO kernel. It cannot be used without its father module CHOCO but it is a separate module so that CHOCO remains small and easier to maintain.

SUGAR is meant to receive any “simple-but-useful” contributions to the OCRE project in order to offer many convenient constraints for developers. Note that major contributions like powerful global constraints are not supposed to be included in SUGAR.

2. Maximum of a list of variables

$V == \max(L)$ holds if V is the maximum of list L .

2.1. Constructor

With $l:\text{list}[\text{IntVar}]$ and $v:\text{IntVar}$:

```
sugar/setMaxConstraint(l,v)
or
v == max(l)
```

2.2. Propagation

This constraint propagates only on bounds. We maintain:

- $v.\text{sup} = \text{Max}\{x.\text{sup} \mid x \text{ in } l\}$
- $v.\text{inf} = \text{Max}\{x.\text{inf} \mid x \text{ in } l\}$
- and when there is only one candidate X to be the max, we propagate $V == X$ (but not holes).

2.3. Examples

Valid situations for $v=\max(\text{list}(A,B,C,D))$:

$V=10$ and $l=(A:5,B:1,C:10,D:0)$

$V=[5..10]$ and $l=(A:[5..6],B:[1..3],C:[2..10],D:0)$

Invalid situations for $v = \max(\text{list}(A,B,C,D))$:

$V=10$ and $l=(A:5,B:1,C:12,D:0)$

$V=[8.10]$ and $l=(A:[2.5],B:[1.3],C:[6.10],D:0)$ //c.inf should be 8 since $V==C$ (only candidate)

3. Product of two variables

$X * Y == Z$ holds if Z is the product of X and Y .

3.1. Constructor

With $x:\text{IntVar}, y:\text{IntVar}, z:\text{IntVar}$:

```
sugar/product(x,y,z) //for x*y=z
```

or

```
x * y == z
```

3.2. Propagation

This constraint propagates only on bounds.

3.3. Examples

Valid situations for $x * y == z$:

$X=2, Y=3$ and $Z=6$

$X=[-2.3], Y=[1.5]$ and $Z=[-10,15]$

Invalid situations for $x * y == z$:

$X=3, Y=3$ and $Z=4$

$X=[-2.3], Y=0$ and $Z=[0.2]$ //Z should be instantiated to 0

4. Element of a list of variables

$V == L[I+c]$ holds if V is equal to the $(I+c)^{\text{th}}$ element of list L .

4.1. Constructor

With $val:\text{IntVar}, idx:\text{IntVar}, l:\text{list}[\text{IntVar}], offset:\text{integer}$:

```
sugar/elt(l,idx,val,offset)
```

or

```
sugar/elt(l,idx,val) //when offset=0
```

or

```
l[idx + offset] == val
```

4.2. Propagation

This constraint propagates only on bounds. We maintain:

- $Val.\text{sup} = \text{Max}\{x.\text{sup} \mid x \text{ in } l\}$
- $Val.\text{inf} = \text{Max}\{x.\text{inf} \mid x \text{ in } l\}$
- And when idx is instantiated, we behave like an equality $val == X$ (where X is $l[idx.\text{value}]$)

4.3. Examples

Valid situations for $l[idx] == val$ (with $l=\text{list}(A,B,C)$)

$Idx=2, val = 3, l=(A:9,B:3,C:0)$

$Idx=[1.2], val = [3.10], l=(A:[5.10],B:[3.8],C:[0.16])$

Invalid situations for $l[idx] == val$ (with $l=\text{list}(A,B,C)$)

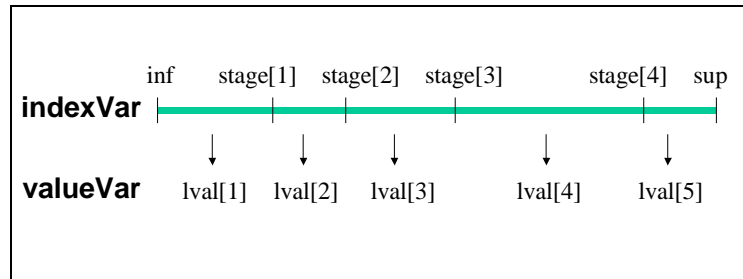
$Idx=2, val = 9, l=(A:9,B:3,C:0)$

$Idx=2, val = [3.10], l=(A:[5.10],B:[3.8],C:[0.16])$ //val should be [3.8] like B since $idx=1$

5. Element with stages

This constraint has the following semantics: $(\text{index} \% (\text{stages}[\text{n}] .. (\text{stages}[\text{n} + 1] - 1)) \Leftrightarrow (\text{value} = \text{lval}[\text{n} + 1])$, (with $\text{stages}[0]=\text{MININT}$ and $\text{stages}[\text{length}(\text{stages}) + 1] = \text{MAXINT}$).

That is to say that the domain of the index is divided in intervals, each of these intervals corresponding to a possible value for the “value” variable.



5.1. Constructor

With $\text{lval}:\text{list}[\text{integer}]$, $\text{stages}:\text{list}[\text{integer}]$, $\text{idx}:\text{IntVar}$, $\text{val}:\text{IntVar}$,

```
sugar/stageElt(lval, stages, idx, val)
```

Where “lval” is the list of possible values, “stages” is the list of stages “idx” is the index variable and “val” is the value variable (ie: $\text{val} == \text{lval}[\text{stageIndex}(\text{idx})]$ if *stageIndex* converts a value of the index variable to an index of list “l”).

5.2. Propagation

This constraint propagates only on bounds.

5.3. Examples

Valid situations for `stageElt(lval, stages, idx, val)` with $\text{lval}=\text{list}(8,5,6)$, $\text{stages}=\text{list}(2,5)$

$\text{Idx}=-4$, $\text{val} = 8$

$\text{Idx}=3$, $\text{val} = 5$,

$\text{Idx}=[-10,4]$, $\text{val} = \{5,8\}$,

Invalid situations for `stageElt(lval, stages, idx, val)` with $\text{lval}=\text{list}(8,5,6)$, $\text{stages}=\text{list}(2,5)$

$\text{val} = 4$

$\text{Idx}=9$, $\text{val} = 5$,

$\text{Idx}=[3,5]$, $\text{val} = [5,6]$, //val should be instantiated to 5

6. Absolute value

$X == |Y|$ holds when X is the absolute value of Y.

6.1. Constructor

With $x:\text{IntVar}$, $y:\text{IntVar}$:

```
sugar/absxy(x, y) //for x = |y|
```

or

```
x == abs(y)
```

6.2. Propagation

The propagation of this constraint is (a priori) complete on enumerated domains (and *a fortiori* on bounds).

6.3. Examples

Valid situations for $x == \text{abs}(y)$

$x=4$, $y = -4$

$x=[2,5]$, $y = \{-3,-2,3,4,5\}$,

Invalid situations for $x == \text{abs}(y)$

$x = -4, y = -4$

$x = \{0,1,2,3\}, y = \{-2,0,2,3\}$ //x should be $\{0,2,3\}$ since -1 and 1 are symmetrical holes

7. Occurrence of a list of value

This constraint is an extension of the choco/occur constraint. The syntax and the propagation are similar.

7.1. Constructor

With $lvar$: $\text{list}[\text{IntVar} \cup \text{BoundIntVar}]$ and $lval$: $\text{list}[\text{integer}] \cup \text{set}[\text{integer}] \cup \text{Interval}$

$\text{sugar/occur}(lval, lvar) \geq 4$

Where occur is the occurrence of lval values in lvar.

The occur method generate an occurTermList object which must be combined with an IntVar or a constant integer. The combination possibilities are the same as in the choco/occur, one can use $==, \geq, <=$.

7.2. Propagation

The propagation of this constraint should be complete on enumerated domains, but is incomplete on bound variables (as the choco/occur).

7.3. Examples

Valid situations for $\text{occur}(\text{list}(x,y), \text{IVal}) == n$

$x = \{5,7,8,9,10\}, y = \{2,3,4\}, n = [0,1], \text{IVal} = \text{list}(3,4,6)$

$x = \{1,3,4\}, y = [1,10], n = [1,2], \text{IVal} = (1 .. 4)$

Invalid situations for $\text{occur}(\text{list}(x,y), \text{IVal}) == n$

$x = \{5,7,8,9,10\}, y = \{3,4\}, n = \{0,2\}, \text{IVal} = \text{list}(3,4,6)$

8. Constraints simplification

8.1. Simplify

This feature is not a constraint but a functionality allowing to automatically simplifying constraints.

The `simplify(c: IntConstraint)` method returns either the constraint itself or an equivalent constraint that is simpler. Particularly it can return a DummyConstraint i.e. either TRUE_C, FALSE_C that have the following semantics:

- `post(TRUE_C)` does nothing.
- `post(FALSE_C)` throws a contradiction.

By now simplify is only implemented for linear constraint. To describe its behavior the linear equation will be denoted:

$$\sum_{i=1}^n a_i X_i + c \geq 0 \quad (\text{or } ==).$$

The simplification depends on the number of significative (with non-nul coefficient) variables:

- If 0, it returns TRUE_C or FALSE_C depending on the value of the c .
- If 1, it is an unary comparison: $X \geq \lceil -c/a \rceil$ (in case of equality, FALSE_C is returned if a does not divide c).
- If 2: If coefficients are not opposite there is no simplification. Otherwise $X_1 \geq X_2 - \lfloor c/a_1 \rfloor$ (here again the equality is impossible if a_1 does not divide c).
- If more, no simplification is possible.

8.2. *disconnectAllEntailed*

A method `disconnectAllEntailed(pb: Problem)` is offered that disconnect all constraint that are entailed. It can be useful to call it after an initial propagation.

9. Shaving

9.1. *lightPropagate*

This method `choco/lightPropagate(p: Problem, maxNbPop: integer, maxQueue: integer)` allows to perform a limited propagation, with limitations of two kind:

- `maxNbPop` specifies the max number of events to pop (i.e. to propagate)
- `maxQueue` disable the propagation of delayed constraints of priority strictly greater than `maxQueue`.

This is especially useful to perform limited shaving.

9.2. *shaveDomain*

This method allows shaving the domain of a variable: for each value of the domain we try to set the variable to this value and then to propagate this event. If a contradiction occurs, the value is removed from the domain.

It is also possible to specify a second variable which bounds will be observed for each value of the first one. Once all values have be tried, the bounds of the second variable can be updated respectively to the maximum of its sup and the minimum of its inf in the different cases. The API is the following:

- `shaveDomain(x: IntVar)` performs the basic shaving
- `shaveDomain(x: IntVar, aff: (IntVarU{unknown}))` performs the shaving and updates the bounds of `aff`.
- `shaveDomain(x: IntVar, aff: (IntVarU{unknown}), maxNbPop: integer, maxQueue: integer)` performs the same shaving using limitations on the propagation (cf. 9.1)

All these methods (including those of 9.2) return `true` when they detect something.

9.3. *shaveBounds*

`ShaveBounds` just tries to eliminate values on the left or on the right of the domain (useful for “`BoundIntVar`”). The API also provide methods to shave only one side of the domain:

- `shaveBounds(x: IntVar)` shaves the bounds of `x`.
- `shaveFromLeft(x: IntVar)` shaves only the inf.
- `shaveFromRight(x: IntVar)` shaves only the sup.

As in 9.2, the `maxNbPop` and `maxQueue` parameters can be added to specify propagation limitations.