org.chocosolver.solver.constraints

Interface IIntConstraintFactory

All Superinterfaces:

ISelf<Model>

All Known Subinterfaces:

IConstraintFactory, IModel

All Known Implementing Classes:

Model

```
public interface IIntConstraintFactory
extends ISelf<Model>
```

Interface to make constraints over BoolVar and IntVar A kind of factory relying on interface default implementation to allow (multiple) inheritance

Since:

4.0.0

Author:

Jean-Guillaume FAGES, Charles Prud'homme

Method Summary

All Methods | Instance Methods | Default Methods

Modifier and Type	Method and Description
default Constraint	absolute(IntVar var1, IntVar var2) Creates an absolute value constraint: var1 = |var2|
default Constraint	allDifferent(IntVar... vars) Creates an allDifferent constraint.
default Constraint	allDifferent(IntVar[] vars, String CONSISTENCY) Creates an allDifferent constraint.
default Constraint	allDifferentExcept0(IntVar[] vars)

	Creates an allDifferent constraint for variables that are not equal to 0.
default Constraint	**allDifferentUnderCondition**(IntVar[] vars, Condition condition, boolean singleCondition) Creates an allDifferent constraint subject to the given condition.
default Constraint	**allEqual**(IntVar... vars) Creates an allEqual constraint.
default Constraint	**among**(IntVar nbVar, IntVar[] vars, int[] values) Creates an among constraint.
default Constraint	**and**(BoolVar... bools) Creates an and constraint that is satisfied if all boolean variables in *bools* are true
default Constraint	**and**(Constraint... cstrs) Creates an and constraint that is satisfied if all constraints in *cstrs* are satisfied BEWARE: this should not be used to post several constraints at once but in a reification context
default Constraint	**arithm**(IntVar var, String op, int cste) Creates an arithmetic constraint : var op cste, where op in {"=", "!
default Constraint	**arithm**(IntVar var1, String op, IntVar var2) Creates an arithmetic constraint: var1 op var2, where op in {"=", "!
default Constraint	**arithm**(IntVar var1, String op1, IntVar var2, String op2, int cste) Creates an arithmetic constraint : var1 op var2, where op in {"=", "!
default Constraint	**arithm**(IntVar var1, String op1, IntVar var2, String op2, IntVar var3) Creates an arithmetic constraint: var1 op1 var2 op2 var3, where op1 and op2 in {"=", "!
default Constraint	**atLeastNValues**(IntVar[] vars, IntVar nValues, boolean AC) Creates an atLeastNValue constraint.
default Constraint	**atMostNValues**(IntVar[] vars, IntVar nValues, boolean STRONG) Creates an atMostNValue constraint.

default Constraint	**binPacking**(IntVar[] itemBin, int[] itemSize, IntVar[] binLoad, int offset) Creates a BinPacking constraint.
default Constraint	**bitsIntChanneling**(BoolVar[] bits, IntVar var) Creates an channeling constraint between an integer variable and a set of bit variables.
default Constraint	**boolsIntChanneling**(BoolVar[] bVars, IntVar var, int offset) Creates an channeling constraint between an integer variable and a set of boolean variables.
default Constraint	**circuit**(IntVar[] vars) Creates a circuit constraint which ensures that the elements of vars define a covering circuit where vars[i] = offset+j means that j is the successor of i.
default Constraint	**circuit**(IntVar[] vars, int offset) Creates a circuit constraint which ensures that the elements of vars define a covering circuit where vars[i] = offset+j means that j is the successor of i.
default Constraint	**circuit**(IntVar[] vars, int offset, CircuitConf conf) Creates a circuit constraint which ensures that the elements of vars define a covering circuit where vars[i] = offset+j means that j is the successor of i.
default Constraint	**clausesIntChanneling**(IntVar var, BoolVar[] eVars, BoolVar[] lVars) Creates an channeling constraint between an integer variable and a set of clauses.
default Constraint	**costRegular**(IntVar[] vars, IntVar cost, ICostAutomaton costAutomaton) Creates a regular constraint that supports a cost function.
default Constraint	**count**(int value, IntVar[] vars, IntVar limit) Creates a count constraint.
default Constraint	

	count(**IntVar** value, **IntVar**[] vars, **IntVar** limit) Creates a count constraint.
default void	**cumulative**(**IntVar**[] starts, int[] durations, int[] heights, int capacity) Creates and **posts** a decomposition of a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
default **Constraint**	**cumulative**(**Task**[] tasks, **IntVar**[] heights, **IntVar** capacity) Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
default **Constraint**	**cumulative**(**Task**[] tasks, **IntVar**[] heights, **IntVar** capacity, boolean incremental) Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
default **Constraint**	**cumulative**(**Task**[] tasks, **IntVar**[] heights, **IntVar** capacity, boolean incremental, **Cumulative.Filter**... filters) Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
default **Constraint**	**cumulative**(**Task**[] tasks, **IntVar**[] heights, **IntVar** capacity, boolean incremental, **CumulFilter**... filters) Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.
default **Constraint**	**diffN**(**IntVar**[] X, **IntVar**[] Y, **IntVar**[] width, **IntVar**[] height, boolean addCumulativeReasoning) Creates a diffN constraint.
default **Constraint**	**distance**(**IntVar** var1, **IntVar** var2, **String** op, int cste) Creates a distance constraint : |var1-var2| op cste where op can take its value among {"=", ">", "<", "!
default **Constraint**	**distance**(**IntVar** var1, **IntVar** var2, **String** op, **IntVar** var3)

	Creates a distance constraint: |var1-var2| op var3 where op can take its value among {"=", ">", "<"}
default Constraint	**div**(IntVar dividend, IntVar divisor, IntVar result) Creates an euclidean division constraint.
default Constraint	**element**(IntVar value, int[] table, IntVar index) Creates an element constraint: value = table[index]
default Constraint	**element**(IntVar value, int[] table, IntVar index, int offset) Creates an element constraint: value = table[index-offset]
default Constraint	**element**(IntVar value, IntVar[] table, IntVar index, int offset) Creates a element constraint: value = table[index-offset] where table is an array of variables.
default int[]	**getDomainUnion**(IntVar... vars) Get the list of values in the domains of vars
default Constraint	**globalCardinality**(IntVar[] vars, int[] values, IntVar[] occurrences, boolean closed) Creates a global cardinality constraint (GCC): Each value values[i] should be taken by exactly occurrences[i] variables of vars.
default Constraint	**intValuePrecedeChain**(IntVar[] X, int[] V) Creates an intValuePrecedeChain constraint.
default Constraint	**intValuePrecedeChain**(IntVar[] X, int S, int T) Creates an intValuePrecedeChain constraint.
default Constraint	**inverseChanneling**(IntVar[] vars1, IntVar[] vars2) Creates an inverse channeling between vars1 and vars2: vars1[i] = j <=> vars2[j] = i Performs AC if domains are enumerated.
default Constraint	**inverseChanneling**(IntVar[] vars1, IntVar[] vars2, int offset1, int offset2) Creates an inverse channeling between vars1 and vars2: vars1[i-offset2] = j <=> vars2[j-offset1] = i Performs AC if domains are enumerated.
default Constraint	**keySort**(IntVar[][] vars, IntVar[] PERMvars,

	IntVar[][] SORTEDvars, int K) Creates a keySort constraint which ensures that the variables of SORTEDvars correspond to the variables of vars according to a permutation stored in PERMvars (optional, can be null).
default **Constraint**	**knapsack**(**IntVar**[] occurrences, **IntVar** weightSum, **IntVar** energySum, int[] weight, int[] energy) Creates a knapsack constraint.
default **Constraint**	**lexChainLess**(**IntVar**[]... vars) Creates a lexChainLess constraint.
default **Constraint**	**lexChainLessEq**(**IntVar**[]... vars) Creates a lexChainLessEq constraint.
default **Constraint**	**lexLess**(**IntVar**[] vars1, **IntVar**[] vars2) Creates a lexLess constraint.
default **Constraint**	**lexLessEq**(**IntVar**[] vars1, **IntVar**[] vars2) Creates a lexLessEq constraint.
default **Constraint**	**max**(**BoolVar** max, **BoolVar**[] vars) Creates a maximum constraint.
default **Constraint**	**max**(**IntVar** max, **IntVar**[] vars) Creates a maximum constraint.
default **Constraint**	**max**(**IntVar** max, **IntVar** var1, **IntVar** var2) Creates a maximum constraint : max = max(var1, var2) (Bound Consistency)
default **Constraint**	**mddc**(**IntVar**[] vars, **MultivaluedDecisionDiagram** MDD) Create a constraint where solutions (tuples) are encoded by a multi-valued decision diagram.
default **Constraint**	**member**(**IntVar** var, int[] table) Creates a member constraint.
default **Constraint**	**member**(**IntVar** var, int lb, int ub) Creates a member constraint.
default **Constraint**	**member**(**IntVar** var, **IntIterableRangeSet** set) Creates a member constraint.
default **Constraint**	**min**(**BoolVar** min, **BoolVar**[] vars) Creates a minimum constraint.

default **Constraint** **min**(**IntVar** min, **IntVar**[] vars)
Creates a minimum constraint.

default **Constraint** **min**(**IntVar** min, **IntVar** var1, **IntVar** var2)
Creates a minimum constraint: min = min(var1, var2) (Bound Consistency)

default **Constraint** **mod**(**IntVar** X, int mod, int res)
Creates a modulo constraint.

default **Constraint** **mod**(**IntVar** X, int mod, **IntVar** Y)
Creates a modulo constraint: X % a = Y

default **Constraint** **mod**(**IntVar** X, **IntVar** Y, **IntVar** Z)
Ensures X % Y = Z.

default **Constraint** **multiCostRegular**(**IntVar**[] vars, **IntVar**[] costVars, **ICostAutomaton** costAutomaton)
Creates a regular constraint that supports a multiple cost function.

default **Constraint** **multiCostRegular**(**IntVar**[] vars, **IntVar**[] costVars, **ICostAutomaton** costAutomaton, double precision)
Creates a regular constraint that supports a multiple cost function.

default **Constraint** **not**(**Constraint** cstr)
Gets the opposite of a given constraint Works for any constraint, including globals, but the associated performances might be weak

default **Constraint** **notAllEqual**(**IntVar**... vars)
Creates a notAllEqual constraint.

default **Constraint** **notMember**(**IntVar** var, int[] table)
Creates a notMember constraint.

default **Constraint** **notMember**(**IntVar** var, int lb, int ub)
Creates a notMember constraint.

default **Constraint** **notMember**(**IntVar** var, **IntIterableRangeSet** set)
Creates a notMember constraint.

default **Constraint** **nValues**(**IntVar**[] vars, **IntVar** nValues)
Creates an nValue constraint.

default Constraint	**or**(BoolVar... bools) Creates an or constraint that is satisfied if at least one boolean variables in *bools* is true
default Constraint	**or**(Constraint... cstrs) Creates an or constraint that is satisfied if at least one constraint in *cstrs* are satisfied
default Constraint	**path**(IntVar[] vars, IntVar start, IntVar end) Creates a path constraint which ensures that the elements of vars define a covering path from start to end where vars[i] = j means that j is the successor of i.
default Constraint	**path**(IntVar[] vars, IntVar start, IntVar end, int offset) Creates a path constraint which ensures that the elements of vars define a covering path from start to end where vars[i] = offset+j means that j is the successor of i.
default Constraint	**regular**(IntVar[] vars, IAutomaton automaton) Creates a regular constraint.
default Constraint	**scalar**(IntVar[] vars, int[] coeffs, String operator, int scalar) Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar
default Constraint	**scalar**(IntVar[] vars, int[] coeffs, String operator, int scalar, int minCardForDecomp) Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar
default Constraint	**scalar**(IntVar[] vars, int[] coeffs, String operator, IntVar scalar) Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar
default Constraint	**scalar**(IntVar[] vars, int[] coeffs, String operator, IntVar scalar, int minCardForDecomp) Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar
default Constraint	**sort**(IntVar[] vars, IntVar[] sortedVars)

Creates a sort constraint which ensures that the variables of sortedVars correspond to the variables of vars according to a permutation.

default **Constraint** **square**(**IntVar** var1, **IntVar** var2)
Creates a square constraint: var1 = var2^2

default **Constraint** **subCircuit**(**IntVar**[] vars, int offset, **IntVar** subCircuitLength)
Creates a subCircuit constraint which ensures that

the elements of vars define a single circuit of subcircuitSize nodes where

vars[i] = offset+j means that j is the successor of i.

default **Constraint** **subPath**(**IntVar**[] vars, **IntVar** start, **IntVar** end, int offset, **IntVar** SIZE)
Creates a subPath constraint which ensures that

the elements of vars define a path of SIZE vertices, leading from start to end

where vars[i] = offset+j means that j is the successor of i.

default **Constraint** **sum**(**BoolVar**[] vars, **String** operator, int sum)
Creates a sum constraint.

default **Constraint** **sum**(**BoolVar**[] vars, **String** operator, **IntVar** sum)
Creates a sum constraint.

default **Constraint** **sum**(**BoolVar**[] vars, **String** operator, **IntVar** sum, int minCardForDecomp)
Creates a sum constraint.

default **Constraint** **sum**(**IntVar**[] vars, **String** operator, int sum)
Creates a sum constraint.

default **Constraint** **sum**(**IntVar**[] vars, **String** operator, int sum, int minCardForDecomp)
Creates a sum constraint.

default **Constraint** **sum**(**IntVar**[] vars, **String** operator, **IntVar** sum)
Creates a sum constraint.

default **Constraint** **sum**(**IntVar**[] vars, **String** operator, **IntVar** sum, int minCardForDecomp)

Creates a sum constraint.

`default` **`Constraint`** **`table`**`(`**`IntVar`**`[] vars, `**`Tuples`**` tuples)`

Creates a table constraint specifying that the sequence of variables vars must belong to the list of tuples (or must NOT belong in case of infeasible tuples) Default configuration with GACSTR+ algorithm for feasible tuples and GAC3rm otherwise

`default` **`Constraint`** **`table`**`(`**`IntVar`**`[] vars, `**`Tuples`**` tuples, `**`String`**` algo)`

Creates a table constraint, with the specified algorithm defined algo - **CT+**: Compact-Table algorithm (AC),
- **GAC2001**: Arc Consistency version 2001 for tuples,
- **GAC2001+**: Arc Consistency version 2001 for allowed tuples,
- **GAC3rm**: Arc Consistency version AC3 rm for tuples,
- **GAC3rm+** (default): Arc Consistency version 3rm for allowed tuples,
- **GACSTR+**: Arc Consistency version STR for allowed tuples,
- **STR2+**: Arc Consistency version STR2 for allowed tuples,
- **FC**: Forward Checking.

`default` **`Constraint`** **`table`**`(`**`IntVar`**` var1, `**`IntVar`**` var2, `**`Tuples`**` tuples)`

Create a table constraint over a couple of variables var1 and var2 Uses AC3rm algorithm by default

`default` **`Constraint`** **`table`**`(`**`IntVar`**` var1, `**`IntVar`**` var2, `**`Tuples`**` tuples, `**`String`**` algo)`

Creates a table constraint over a couple of variables var1 and var2:
- **AC2001**: table constraint which applies the AC2001 algorithm,
- **AC3**: table constraint which applies the AC3 algorithm,
- **AC3rm**: table constraint which applies the AC3 rm algorithm,
- **AC3bit+rm** (default): table constraint which applies the AC3 bit+rm algorithm,
- **FC**: table constraint which applies forward checking algorithm.

`default` **`Constraint`** **`times`**`(`**`IntVar`**` X, int Y, `**`IntVar`**` Z)`

Creates a multiplication constraint: X * Y = Z

`default` **`Constraint`** **`times`**`(`**`IntVar`**` X, `**`IntVar`**` Y, int Z)`

Creates a multiplication constraint: X * Y = Z

default Constraint	**times**(IntVar X, IntVar Y, IntVar Z) Creates a multiplication constraint: X * Y = Z
default Constraint	**tree**(IntVar[] succs, IntVar nbTrees) Creates a tree constraint.
default Constraint	**tree**(IntVar[] succs, IntVar nbTrees, int offset) Creates a tree constraint.

Methods inherited from interface org.chocosolver.solver.ISelf

ref

Method Detail

arithm

```
default Constraint arithm(IntVar var,
                          String op,
                          int cste)
```

Creates an arithmetic constraint : var op cste, where op in {"=", "!=", ">","<",">=","<="}

Parameters:

var - a variable

op - an operator

cste - a constant

member

```
default Constraint member(IntVar var,
                          int[] table)
```

Creates a member constraint. Ensures var takes its values in table

Parameters:

var - an integer variable

table - an array of values

member

```
default Constraint member(IntVar var,
                          int lb,
                          int ub)
```

Creates a member constraint. Ensures var takes its values in [LB, UB]

Parameters:

`var - an integer variable`

`lb - the lower bound of the interval`

`ub - the upper bound of the interval`

mod

```
default Constraint mod(IntVar X,
                       int mod,
                       int res)
```

Creates a modulo constraint. Ensures X % a = b

Parameters:

`X - an integer variable`

`mod - the value of the modulo operand`

`res - the result of the modulo operation`

not

```
default Constraint not(Constraint cstr)
```

Gets the opposite of a given constraint Works for any constraint, including globals, but the associated performances might be weak

Parameters:

`cstr - a constraint`

Returns:

`the opposite constraint of` *`cstr`*

notMember

```
default Constraint notMember(IntVar var,
                             int[] table)
```

Creates a notMember constraint. Ensures var does not take its values in table

Parameters:

`var - an integer variable`

`table - an array of values`

member

```
default Constraint member(IntVar var,
                          IntIterableRangeSet set)
```

Creates a member constraint. Ensures var takes its values in set

Parameters:

`var - an integer variable`

`set - a set of values`

notMember

```
default Constraint notMember(IntVar var,
                             int lb,
                             int ub)
```

Creates a notMember constraint. Ensures var does not take its values in [lb, UB]

Parameters:

`var - an integer variable`

`lb - the lower bound of the interval`

`ub - the upper bound of the interval`

notMember

```
default Constraint notMember(IntVar var,
                             IntIterableRangeSet set)
```

Creates a notMember constraint. Ensures var does not take its values in set

Parameters:

`var - an integer variable`

`set - a set of values`

absolute

```
default Constraint absolute(IntVar var1,
                            IntVar var2)
```

Creates an absolute value constraint: var1 = |var2|

arithm

```
default Constraint arithm(IntVar var1,
                          String op,
                          IntVar var2)
```

Creates an arithmetic constraint: var1 op var2, where op in {"=", "!=", ">","<",">=","<="}

Parameters:

var1 - first variable

op - an operator

var2 - second variable

arithm

```
default Constraint arithm(IntVar var1,
                          String op1,
                          IntVar var2,
                          String op2,
                          int cste)
```

Creates an arithmetic constraint : var1 op var2, where op in {"=", "!=", ">","<",">=","<="} or {"+", "-", "*", "/"}

Parameters:

var1 - first variable

op1 - an operator

var2 - second variable

op2 - another operator

cste - an operator

distance

```
default Constraint distance(IntVar var1,
                            IntVar var2,
                            String op,
                            int cste)
```

Creates a distance constraint : |var1-var2| op cste
where op can take its value among {"=", ">", "<", "!="}

element

```
default Constraint element(IntVar value,
                           int[] table,
                           IntVar index,
                           int offset)
```

Creates an element constraint: value = table[index-offset]

Parameters:

`value - an integer variable taking its value in table`

`table - an array of integer values`

`index - an integer variable representing the value of value in table`

`offset - offset matching index.lb and table[0] (Generally 0)`

element

```
default Constraint element(IntVar value,
                           int[] table,
                           IntVar index)
```

Creates an element constraint: value = table[index]

Parameters:

`value - an integer variable taking its value in table`

`table - an array of integer values`

`index - an integer variable representing the value of value in table`

mod

```
default Constraint mod(IntVar X,
                       int mod,
                       IntVar Y)
```

Creates a modulo constraint: X % a = Y

Parameters:

`X - first integer variable`

mod - the value of the modulo operand

Y - second integer variable (result of the modulo operation)

square

```
default Constraint square(IntVar var1,
                          IntVar var2)
```

Creates a square constraint: var1 = var2^2

table

```
default Constraint table(IntVar var1,
                         IntVar var2,
                         Tuples tuples)
```

Create a table constraint over a couple of variables var1 and var2 Uses AC3rm algorithm by default

Parameters:

var1 - first variable

var2 - second variable

table

```
default Constraint table(IntVar var1,
                         IntVar var2,
                         Tuples tuples,
                         String algo)
```

Creates a table constraint over a couple of variables var1 and var2:
- **AC2001**: table constraint which applies the AC2001 algorithm,
- **AC3**: table constraint which applies the AC3 algorithm,
- **AC3rm**: table constraint which applies the AC3 rm algorithm,
- **AC3bit+rm** (default): table constraint which applies the AC3 bit+rm algorithm,
- **FC**: table constraint which applies forward checking algorithm.

Parameters:

var1 - first variable

var2 - second variable

tuples - the relation between the two variables, among {"AC3", "AC3rm", "AC3bit+rm", "AC2001", "FC"}

times

```
default Constraint times(IntVar X,
                         int Y,
                         IntVar Z)
```

Creates a multiplication constraint: X * Y = Z

Parameters:

`X - first variable`

`Y - a constant`

`Z - result variable`

times

```
default Constraint times(IntVar X,
                         IntVar Y,
                         int Z)
```

Creates a multiplication constraint: X * Y = Z

Parameters:

`X - first variable`

`Y - second variable`

`Z - a constant (result)`

arithm

```
default Constraint arithm(IntVar var1,
                          String op1,
                          IntVar var2,
                          String op2,
                          IntVar var3)
```

Creates an arithmetic constraint: var1 op1 var2 op2 var3, where op1 and op2 in {"=", "!=", ">","<",">=","<="} or {"+", "-", "*", "/"}

Parameters:

`var1 - first variable`

`op1 - an operator`

`var2 - second variable`

`op2 - another operator`

var3 - third variable

distance

```
default Constraint distance(IntVar var1,
                            IntVar var2,
                            String op,
                            IntVar var3)
```

Creates a distance constraint: |var1-var2| op var3
where op can take its value among {"=", ">", "<"}

Parameters:

var1 - first variable

var2 - second variable

op - an operator

var3 - resulting variable

div

```
default Constraint div(IntVar dividend,
                       IntVar divisor,
                       IntVar result)
```

Creates an euclidean division constraint. Ensures dividend / divisor = result, rounding towards 0 Also ensures divisor != 0

Parameters:

dividend - dividend

divisor - divisor

result - result

max

```
default Constraint max(IntVar max,
                       IntVar var1,
                       IntVar var2)
```

Creates a maximum constraint : max = max(var1, var2) (Bound Consistency)

Parameters:

max - a variable

var1 - a variable

var2 - a variable

min

```
default Constraint min(IntVar min,
                       IntVar var1,
                       IntVar var2)
```

Creates a minimum constraint: min = min(var1, var2) (Bound Consistency)

Parameters:

min - a variable

var1 - a variable

var2 - a variable

mod

```
default Constraint mod(IntVar X,
                       IntVar Y,
                       IntVar Z)
```

Ensures X % Y = Z.

Creates a modulo constraint, that uses truncated division: the quotient is defined by truncation q = trunc(a/n) and the remainder would have same sign as the dividend. The quotient is rounded towards zero: equal to the first integer in the direction of zero from the exact rational quotient.

Parameters:

X - first variable

Y - second variable

Z - result

times

```
default Constraint times(IntVar X,
                         IntVar Y,
                         IntVar Z)
```

Creates a multiplication constraint: X * Y = Z

Parameters:

X - first variable

Y - second variable

Z - result variable

allDifferent

```
default Constraint allDifferent(IntVar... vars)
```

Creates an allDifferent constraint. Ensures that all variables from vars take a different value. Uses BC plus a probabilistic AC propagator to get a compromise between BC and AC

Parameters:

vars - list of variables

allDifferent

```
default Constraint allDifferent(IntVar[] vars,
                                String CONSISTENCY)
```

Creates an allDifferent constraint. Ensures that all variables from vars take a different value. The consistency level should be chosen among "BC", "AC" and "DEFAULT".

Parameters:

vars - list of variables

CONSISTENCY - consistency level, among {"BC", "AC"}

BC: Based on: "A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint"
A. Lopez-Ortiz, CG. Quimper, J. Tromp, P.van Beek
AC: Uses Regin algorithm Runs in O(m.n) worst case time for the initial propagation and then in O(n+m) on average.

DEFAULT:
Uses BC plus a probabilistic AC propagator to get a compromise between BC and AC

allDifferentUnderCondition

```
default Constraint allDifferentUnderCondition(IntVar[] vars,
                                              Condition condition,
                                              boolean singleCondition)
```

Creates an allDifferent constraint subject to the given condition. More precisely: IF

singleCondition for all X,Y in vars, condition(X) => X != Y ELSE for all X,Y in vars, condition(X) AND condition(Y) => X != Y

Parameters:

vars - collection of variables

condition - condition defining which variables should be constrained

singleCondition - specifies how to apply filtering

allDifferentExcept0

```
default Constraint allDifferentExcept0(IntVar[] vars)
```

Creates an allDifferent constraint for variables that are not equal to 0. There can be multiple variables equal to 0.

Parameters:

vars - collection of variables

allEqual

```
default Constraint allEqual(IntVar... vars)
```

Creates an allEqual constraint. Ensures that all variables from vars take the same value.

Parameters:

vars - list of variables

notAllEqual

```
default Constraint notAllEqual(IntVar... vars)
```

Creates a notAllEqual constraint. Ensures that all variables from vars take more than a single value.

Parameters:

vars - list of variables

among

```
default Constraint among(IntVar nbVar,
                         IntVar[] vars,
```

```
                                int[] values)
```

Creates an among constraint. nbVar is the number of variables of the collection vars that take their value in values.
gccat among
Propagator : C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, T. Walsh, Among, common and disjoint Constraints CP-2005

Parameters:

`nbVar - a variable`

`vars - vector of variables`

`values - set of values`

and

```
default Constraint and(BoolVar... bools)
```

Creates an and constraint that is satisfied if all boolean variables in *bools* are true

Parameters:

`bools - an array of boolean variable`

Returns:

`a constraint and ensuring that variables in bools are all set to true`

and

```
default Constraint and(Constraint... cstrs)
```

Creates an and constraint that is satisfied if all constraints in *cstrs* are satisfied
BEWARE: this should not be used to post several constraints at once but in a reification context

Parameters:

`cstrs - an array of constraints`

Returns:

`a constraint and ensuring that all constraints in cstrs are satisfied`

atLeastNValues

```
default Constraint atLeastNValues(IntVar[] vars,
                                  IntVar nValues,
```

```
                                boolean AC)
```

Creates an atLeastNValue constraint. Let N be the number of distinct values assigned to the variables of the vars collection. Enforce condition N >= nValues to hold.

This embeds a light propagator by default. Additional filtering algorithms can be added.

Parameters:

`vars - collection of variables`

`nValues - limit variable`

`AC - additional filtering algorithm, domain filtering algorithm derivated from (Soft)AllDifferent`

atMostNValues

```
default Constraint atMostNValues(IntVar[] vars,
                                 IntVar nValues,
                                 boolean STRONG)
```

Creates an atMostNValue constraint. Let N be the number of distinct values assigned to the variables of the vars collection. Enforce condition N <= nValues to hold.

This embeds a light propagator by default. Additional filtering algorithms can be added.

Parameters:

`vars - collection of variables`

`nValues - limit variable`

`STRONG - "AMNV" Filters the conjunction of AtMostNValue and disequalities (see Fages and Lapègue Artificial Intelligence 2014) automatically detects disequalities and allDifferent constraints. Presumably useful when nValues must be minimized.`

binPacking

```
default Constraint binPacking(IntVar[] itemBin,
                              int[] itemSize,
                              IntVar[] binLoad,
                              int offset)
```

Creates a BinPacking constraint. Bin Packing formulation: forall b in [0,binLoad.length-1], binLoad[b]=sum(itemSize[i] | i in [0,itemSize.length-1], itemBin[i] = b+offset forall i in [0,itemSize.length-1], itemBin is in [offset,binLoad.length-1+offset],

Parameters:

itemBin - IntVar representing the bin of each item

itemSize - int representing the size of each item

binLoad - IntVar representing the load of each bin (i.e. the sum of the size of the items in it)

offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)

boolsIntChanneling

```
default Constraint boolsIntChanneling(BoolVar[] bVars,
                                      IntVar var,
                                      int offset)
```

Creates an channeling constraint between an integer variable and a set of boolean variables. Maps the boolean assignments variables bVars with the standard assignment variable var.
var = i <-> bVars[i-offset] = 1

Parameters:

bVars - array of boolean variables

var - observed variable. Should presumably have an enumerated domain

offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)

bitsIntChanneling

```
default Constraint bitsIntChanneling(BoolVar[] bits,
                                     IntVar var)
```

Creates an channeling constraint between an integer variable and a set of bit variables. Ensures that var = 2^0*BIT_1 + 2^1*BIT_2 + ... 2^{n-1}*BIT_n.
BIT_1 is related to the first bit of OCTET (2^0), BIT_2 is related to the first bit of OCTET (2^1), etc.
The upper bound of var is given by 2^n, where n is the size of the array bits.

Parameters:

bits - the array of bits

var - the numeric value

clausesIntChanneling

```
default Constraint clausesIntChanneling(IntVar var,
                                        BoolVar[] eVars,
                                        BoolVar[] lVars)
```

Creates an channeling constraint between an integer variable and a set of clauses. Link each value from the domain of var to two boolean variable: one reifies the equality to the i^th value of the variable domain, the other reifies the less-or-equality to the i^th value of the variable domain. Contract: eVars.lenght == lVars.length == var.getUB() - var.getLB() + 1 Contract: var is not a boolean variable

Parameters:

`var - an Integer variable`

`eVars - array of EQ boolean variables`

`lVars - array of LQ boolean variables`

circuit

```
default Constraint circuit(IntVar[] vars)
```

Creates a circuit constraint which ensures that

the elements of vars define a covering circuit

where vars[i] = offset+j means that j is the successor of i.

Filtering algorithms:

subtour elimination : Caseau & Laburthe (ICLP'97)

allDifferent GAC algorithm: Régin (AAAI'94)

dominator-based filtering: Fages & Lorca (CP'11)

Strongly Connected Components based filtering (Cambazar & Bourreau JFPC'06 and Fages and Lorca TechReport'12)

Parameters:

`vars - vector of variables which take their value in [offset,offset+|vars|-1]`

Returns:

`a circuit constraint`

circuit

```
default Constraint circuit(IntVar[] vars,
```

```
                            int offset)
```

Creates a circuit constraint which ensures that

the elements of vars define a covering circuit

where vars[i] = offset+j means that j is the successor of i.

Filtering algorithms:

subtour elimination : Caseau & Laburthe (ICLP'97)

allDifferent GAC algorithm: Régin (AAAI'94)

dominator-based filtering: Fages & Lorca (CP'11)

Strongly Connected Components based filtering (Cambazar & Bourreau JFPC'06 and Fages and Lorca TechReport'12)

Parameters:

`vars - vector of variables which take their value in [offset,offset+|vars|-1]`

`offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)`

Returns:

`a circuit constraint`

circuit

```
default Constraint circuit(IntVar[] vars,
                           int offset,
                           CircuitConf conf)
```

Creates a circuit constraint which ensures that

the elements of vars define a covering circuit

where vars[i] = offset+j means that j is the successor of i.

Filtering algorithms:

subtour elimination : Caseau & Laburthe (ICLP'97)

allDifferent GAC algorithm: Régin (AAAI'94)

dominator-based filtering: Fages & Lorca (CP'11)

Strongly Connected Components based filtering (Cambazard & Bourreau JFPC'06 and Fages and Lorca TechReport'12)

See Fages PhD Thesis (2014) for more information

Parameters:

vars - vector of variables which take their value in [offset,offset+|vars|-1]

offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)

conf - filtering options

Returns:

a circuit constraint

costRegular

```
default Constraint costRegular(IntVar[] vars,
                               IntVar cost,
                               ICostAutomaton costAutomaton)
```

Creates a regular constraint that supports a cost function. Ensures that the assignment of a sequence of variables is recognized by costAutomaton, a deterministic finite automaton, and that the sum of the costs associated to each assignment is bounded by the cost variable. This version allows to specify different costs according to the automaton state at which the assignment occurs (i.e. the transition starts)

Parameters:

vars - sequence of variables

cost - cost variable

costAutomaton - a deterministic finite automaton defining the regular language and the costs Can be built with method CostAutomaton.makeSingleResource(...)

count

```
default Constraint count(int value,
                         IntVar[] vars,
                         IntVar limit)
```

Creates a count constraint. Let N be the number of variables of the vars collection assigned to value value; Enforce condition N = limit to hold.

Parameters:

value - an int

vars - a vector of variables

limit - a variable

count

```
default Constraint count(IntVar value,
                         IntVar[] vars,
                         IntVar limit)
```

Creates a count constraint. Let N be the number of variables of the vars collection assigned to value value; Enforce condition N = limit to hold.

Parameters:

`value - a variable`

`vars - a vector of variables`

`limit - a variable`

cumulative

```
default Constraint cumulative(Task[] tasks,
                              IntVar[] heights,
                              IntVar capacity)
```

Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit. Task duration and height should be >= 0 Discards tasks whose duration or height is equal to zero

Parameters:

`tasks - Task objects containing start, duration and end variables`

`heights - integer variables representing the resource consumption of each task`

`capacity - integer variable representing the resource capacity`

Returns:

`a cumulative constraint`

cumulative

```
default Constraint cumulative(Task[] tasks,
                              IntVar[] heights,
                              IntVar capacity,
                              boolean incremental)
```

Creates a cumulative constraint: Enforces that at each point in time, the cumulated

height of the set of tasks that overlap that point does not exceed a given limit. Task duration and height should be >= 0 Discards tasks whose duration or height is equal to zero

Parameters:

`tasks - Task objects containing start, duration and end variables`

`heights - integer variables representing the resource consumption of each task`

`capacity - integer variable representing the resource capacity`

`incremental - specifies if an incremental propagation should be applied`

Returns:

`a cumulative constraint`

cumulative

```
default Constraint cumulative(Task[] tasks,
                              IntVar[] heights,
                              IntVar capacity,
                              boolean incremental,
                              Cumulative.Filter... filters)
```

Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit. Task duration and height should be >= 0 Discards tasks whose duration or height is equal to zero

Parameters:

`tasks - Task objects containing start, duration and end variables`

`heights - integer variables representing the resource consumption of each task`

`capacity - integer variable representing the resource capacity`

`incremental - specifies if an incremental propagation should be applied`

`filters - specifies which filtering algorithms to apply`

Returns:

`a cumulative constraint`

cumulative

```
default Constraint cumulative(Task[] tasks,
```

```
                                IntVar[] heights,
                                IntVar capacity,
                                boolean incremental,
                                CumulFilter... filters)
```

Creates a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit. Task duration and height should be >= 0 Discards tasks whose duration or height is equal to zero

Parameters:

`tasks - Task objects containing start, duration and end variables`

`heights - integer variables representing the resource consumption of each task`

`capacity - integer variable representing the resource capacity`

`incremental - specifies if an incremental propagation should be applied`

`filters - specifies which filtering algorithms to apply`

Returns:

`a cumulative constraint`

cumulative

```
default void cumulative(IntVar[] starts,
                        int[] durations,
                        int[] heights,
                        int capacity)
```

Creates and **posts** a decomposition of a cumulative constraint: Enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit. Task duration and height should be >= 0 Discards tasks whose duration or height is equal to zero

Parameters:

`starts - starting time of each task`

`durations - processing time of each task`

`heights - resource consumption of each task`

`capacity - resource capacity`

diffN

```
default Constraint diffN(IntVar[] X,
```

```
                            IntVar[] Y,
                            IntVar[] width,
                            IntVar[] height,
                            boolean addCumulativeReasoning)
```

Creates a diffN constraint. Constrains each rectangle$_i$, given by their origins X_i,Y_i and sizes width$_i$,height$_i$, to be non-overlapping.

Parameters:

`X - collection of coordinates in first dimension`

`Y - collection of coordinates in second dimension`

`width - collection of width (each duration should be > 0)`

`height - collection of height (each height should be >= 0)`

`addCumulativeReasoning - indicates whether or not redundant cumulative constraints should be put on each dimension (advised)`

Returns:

`a non-overlapping constraint`

element

```
default Constraint element(IntVar value,
                           IntVar[] table,
                           IntVar index,
                           int offset)
```

Creates a element constraint: value = table[index-offset] where table is an array of variables.

Parameters:

`value - value variable`

`table - array of variables`

`index - index variable in range [offset,offset+|table|-1]`

`offset - int offset, generally 0`

globalCardinality

```
default Constraint globalCardinality(IntVar[] vars,
                                     int[] values,
                                     IntVar[] occurrences,
                                     boolean closed)
```

Creates a global cardinality constraint (GCC): Each value values[i] should be taken by

exactly occurrences[i] variables of vars.
This constraint does not ensure any well-defined level of consistency, yet.

Parameters:

`vars - collection of variables`

`values - collection of constrained values`

`occurrences - collection of cardinality variables`

`closed - restricts domains of vars to values if set to true`

inverseChanneling

```
default Constraint inverseChanneling(IntVar[] vars1,
                                     IntVar[] vars2)
```

Creates an inverse channeling between vars1 and vars2: vars1[i] = j <=> vars2[j] = i Performs AC if domains are enumerated. If not, then it works on bounds without guaranteeing BC (enumerated domains are strongly recommended)

Beware you should have |vars1| = |vars2|

Parameters:

`vars1 - vector of variables which take their value in [0,|vars2|-1]`

`vars2 - vector of variables which take their value in [0,|vars1|-1]`

inverseChanneling

```
default Constraint inverseChanneling(IntVar[] vars1,
                                     IntVar[] vars2,
                                     int offset1,
                                     int offset2)
```

Creates an inverse channeling between vars1 and vars2: vars1[i-offset2] = j <=> vars2[j-offset1] = i Performs AC if domains are enumerated. If not, then it works on bounds without guaranteeing BC (enumerated domains are strongly recommended)

Beware you should have |vars1| = |vars2|

Parameters:

`vars1 - vector of variables which take their value in [offset1,offset1+|vars2|-1]`

`vars2 - vector of variables which take their value in [offset2,offset2+|vars1|-1]`

offset1 - lowest value in vars1 (most often 0)

offset2 - lowest value in vars2 (most often 0)

intValuePrecedeChain

```
default Constraint intValuePrecedeChain(IntVar[] X,
                                        int S,
                                        int T)
```

Creates an intValuePrecedeChain constraint. Ensure that if there exists `j` such that X[j] = T, then, there must exist `i` < `j` such that X[i] = S.

Parameters:

X - an array of variables

S - a value

T - another value

intValuePrecedeChain

```
default Constraint intValuePrecedeChain(IntVar[] X,
                                        int[] V)
```

Creates an intValuePrecedeChain constraint. Ensure that, for each pair of V[k] and V[l] of values in V, such that k < l, if there exists `j` such that X[j] = V[l], then, there must exist `i` < `j` such that X[i] = V[k].

Parameters:

X - array of variables

V - array of (distinct) values

knapsack

```
default Constraint knapsack(IntVar[] occurrences,
                            IntVar weightSum,
                            IntVar energySum,
                            int[] weight,
                            int[] energy)
```

Creates a knapsack constraint. Ensures that :
- occurrences[i] * weight[i] = weightSum
- occurrences[i] * energy[i] = energySum
and maximizing the value of energySum.

A knapsack constraint wikipedia:
"Given a set of items, each with a weight and an energy value, determine the count of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items." The limit over weightSum has to be specified either in its domain or with an additional constraint:

```
model.post(solver.arithm(weightSum, "<=", limit);
```

Parameters:

`occurrences - number of occurrences of every item`

`weightSum - load of the knapsack`

`energySum - profit of the knapsack`

`weight - weight of each item (must be >=0)`

`energy - energy of each item (must be >=0)`

keySort

```
default Constraint keySort(IntVar[][] vars,
                           IntVar[] PERMvars,
                           IntVar[][] SORTEDvars,
                           int K)
```

Creates a keySort constraint which ensures that the variables of SORTEDvars correspond to the variables of vars according to a permutation stored in PERMvars (optional, can be null). The variables of SORTEDvars are also sorted in increasing order wrt to K-size tuples. The sort is stable, that is, ties are broken using the position of the tuple in vars.

For example:
- vars= (<4,2,2>,<2,3,1>,<4,2,1><1,3,0>)
- SORTEDvars= (<1,3,0>,<2,3,1>,<4,2,2>,<4,2,1>)
- PERMvars= (2,1,3,0)
- K = 2

Parameters:

`vars - a tuple of array of variables`

`PERMvars - array of permutation variables, domains should be [1,vars.length] -- Can be null`

`SORTEDvars - a tuple of array of variables sorted in increasing order`

`K - key perfixes size (0 ≤ k ≤ m, where m is the size of the`

array of variable)

Returns:

a keySort constraint

lexChainLess

```
default Constraint lexChainLess(IntVar[]... vars)
```

Creates a lexChainLess constraint. For each pair of consecutive vectors $vars_i$ and $vars_{i+1}$ of the vars collection $vars_i$ is lexicographically strictly less than than $vars_{i+1}$

Parameters:

vars - collection of vectors of variables

lexChainLessEq

```
default Constraint lexChainLessEq(IntVar[]... vars)
```

Creates a lexChainLessEq constraint. For each pair of consecutive vectors $vars_i$ and $vars_{i+1}$ of the vars collection $vars_i$ is lexicographically less or equal than than $vars_{i+1}$

Parameters:

vars - collection of vectors of variables

lexLess

```
default Constraint lexLess(IntVar[] vars1,
                           IntVar[] vars2)
```

Creates a lexLess constraint. Ensures that vars1 is lexicographically strictly less than vars2.

Parameters:

vars1 - vector of variables

vars2 - vector of variables

lexLessEq

```
default Constraint lexLessEq(IntVar[] vars1,
                             IntVar[] vars2)
```

Creates a lexLessEq constraint. Ensures that vars1 is lexicographically less or equal than vars2.

Parameters:

`vars1 - vector of variables`

`vars2 - vector of variables`

max

```
default Constraint max(IntVar max,
                       IntVar[] vars)
```

Creates a maximum constraint. max is the maximum value of the collection of domain variables vars

Parameters:

`max - a variable`

`vars - a vector of variables, of size > 0`

max

```
default Constraint max(BoolVar max,
                       BoolVar[] vars)
```

Creates a maximum constraint. max is the maximum value of the collection of boolean variables vars

Parameters:

`max - a boolean variable`

`vars - a vector of boolean variables, of size > 0`

mddc

```
default Constraint mddc(IntVar[] vars,
                        MultivaluedDecisionDiagram MDD)
```

Create a constraint where solutions (tuples) are encoded by a multi-valued decision diagram. The order of the variables in vars is important and must refer to the MDD.

Parameters:

`vars - the array of variables`

`MDD - the multi-valued decision diagram encoding solutions`

min

```
default Constraint min(IntVar min,
                       IntVar[] vars)
```

Creates a minimum constraint. min is the minimum value of the collection of domain variables vars

Parameters:

`min - a variable`

`vars - a vector of variables, of size > 0`

min

```
default Constraint min(BoolVar min,
                       BoolVar[] vars)
```

Creates a minimum constraint. min is the minimum value of the collection of boolean variables vars

Parameters:

`min - a boolean variable`

`vars - a vector of boolean variables, of size > 0`

multiCostRegular

```
default Constraint multiCostRegular(IntVar[] vars,
                                    IntVar[] costVars,
                                    ICostAutomaton costAutomaton)
```

Creates a regular constraint that supports a multiple cost function. Ensures that the assignment of a sequence of vars is recognized by costAutomaton, a deterministic finite automaton, and that the sum of the cost vector associated to each assignment is bounded by the variable vector costVars. This version allows to specify different costs according to the automaton state at which the assignment occurs (i.e. the transition starts)

Parameters:

`vars - sequence of variables`

`costVars - cost variables`

`costAutomaton - a deterministic finite automaton defining the regular language and the costs Can be built from method CostAutomaton.makeMultiResources(...)`

multiCostRegular

```
default Constraint multiCostRegular(IntVar[] vars,
                                    IntVar[] costVars,
                                    ICostAutomaton costAutomaton,
                                    double precision)
```

Creates a regular constraint that supports a multiple cost function. Ensures that the assignment of a sequence of vars is recognized by costAutomaton, a deterministic finite automaton, and that the sum of the cost vector associated to each assignment is bounded by the variable vector costVars. This version allows to specify different costs according to the automaton state at which the assignment occurs (i.e. the transition starts)

Parameters:

`vars - sequence of variables`

`costVars - cost variables`

`costAutomaton - a deterministic finite automaton defining the regular language and the costs Can be built from method CostAutomaton.makeMultiResources(...)`

`precision - the smallest used double for MCR algorithm`

nValues

```
default Constraint nValues(IntVar[] vars,
                           IntVar nValues)
```

Creates an nValue constraint. Let N be the number of distinct values assigned to the variables of the vars collection. Enforce condition N = nValues to hold.

This embeds a light propagator by default. Additional filtering algorithms can be added.

see atleast_nvalue and atmost_nvalue

Parameters:

`vars - collection of variables`

`nValues - limit variable`

Returns:

`the conjunction of atleast_nvalue and atmost_nvalue`

or

```
default Constraint or(BoolVar... bools)
```

Creates an or constraint that is satisfied if at least one boolean variables in *bools* is true

Parameters:

`bools - an array of boolean variable`

Returns:

`a constraint that is satisfied if at least one boolean variables in bools is true`

or

```
default Constraint or(Constraint... cstrs)
```

Creates an or constraint that is satisfied if at least one constraint in *cstrs* are satisfied

Parameters:

`cstrs - an array of constraints`

Returns:

`a constraint and ensuring that at least one constraint in cstrs are satisfied`

path

```
default Constraint path(IntVar[] vars,
                        IntVar start,
                        IntVar end)
```

Creates a path constraint which ensures that

the elements of vars define a covering path from start to end

where vars[i] = j means that j is the successor of i.

Moreover, vars[end] = |vars|

Requires : |vars|>0

Filtering algorithms: see circuit constraint

Parameters:

`vars - vector of variables which take their value in [0,|vars|]`

`start - variable indicating the index of the first variable in the path`

`end - variable indicating the index of the last variable in the path`

Returns:

`a path constraint`

path

```
default Constraint path(IntVar[] vars,
                        IntVar start,
                        IntVar end,
                        int offset)
```

Creates a path constraint which ensures that

the elements of vars define a covering path from start to end

where vars[i] = offset+j means that j is the successor of i.

Moreover, vars[end-offset] = |vars|+offset

Requires : |vars|>0

Filtering algorithms: see circuit constraint

Parameters:

vars - vector of variables which take their value in [offset,offset+|vars|]

start - variable indicating the index of the first variable in the path

end - variable indicating the index of the last variable in the path

offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)

Returns:

a path constraint

regular

```
default Constraint regular(IntVar[] vars,
                           IAutomaton automaton)
```

Creates a regular constraint. Enforces the sequence of vars to be a word recognized by the deterministic finite automaton. For example regexp = "(1|2)(3*)(4|5)"; The same dfa can be used for different propagators.

Parameters:

vars - sequence of variables

automaton - a deterministic finite automaton defining the regular language

scalar

```
default Constraint scalar(IntVar[] vars,
                          int[] coeffs,
                          String operator,
                          int scalar)
```

Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar

Parameters:

`vars - a collection of IntVar`

`coeffs - a collection of int, for which |vars|=|coeffs|`

`operator - an operator in {"=", "!=", ">","<",">=","<="}`

`scalar - an integer`

Returns:

`a scalar constraint`

scalar

```
default Constraint scalar(IntVar[] vars,
                          int[] coeffs,
                          String operator,
                          int scalar,
                          int minCardForDecomp)
```

Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar

Parameters:

`vars - a collection of IntVar`

`coeffs - a collection of int, for which |vars|=|coeffs|`

`operator - an operator in {"=", "!=", ">","<",">=","<="}`

`scalar - an integer`

`minCardForDecomp - minimum number of cardinality threshold to a sum constraint to be decomposed`

Returns:

`a scalar constraint`

scalar

```
default Constraint scalar(IntVar[] vars,
                          int[] coeffs,
```

```
                          String operator,
                          IntVar scalar)
```

Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar

Parameters:

`vars - a collection of IntVar`

`coeffs - a collection of int, for which |vars|=|coeffs|`

`operator - an operator in {"=", "!=", ">","<",">=","<="}`

`scalar - an IntVar`

Returns:

`a scalar constraint`

scalar

```
default Constraint scalar(IntVar[] vars,
                          int[] coeffs,
                          String operator,
                          IntVar scalar,
                          int minCardForDecomp)
```

Creates a scalar constraint which ensures that Sum(vars[i]*coeffs[i]) operator scalar

Parameters:

`vars - a collection of IntVar`

`coeffs - a collection of int, for which |vars|=|coeffs|`

`operator - an operator in {"=", "!=", ">","<",">=","<="}`

`scalar - an IntVar`

`minCardForDecomp - minimum number of cardinality threshold to a sum constraint to be decomposed`

Returns:

`a scalar constraint`

sort

```
default Constraint sort(IntVar[] vars,
                        IntVar[] sortedVars)
```

Creates a sort constraint which ensures that the variables of sortedVars correspond to the variables of vars according to a permutation. The variables of sortedVars are also sorted in increasing order.

For example:
- X= (4,2,1,3)
- Y= (1,2,3,4)

Parameters:

`vars - an array of variables`

`sortedVars - an array of variables sorted in increasing order`

Returns:

`a sort constraint`

subCircuit

```
default Constraint subCircuit(IntVar[] vars,
                              int offset,
                              IntVar subCircuitLength)
```

Creates a subCircuit constraint which ensures that

the elements of vars define a single circuit of subcircuitSize nodes where

vars[i] = offset+j means that j is the successor of i.

and vars[i] = offset+i means that i is not part of the circuit

the constraint ensures that |{vars[i] =/= offset+i}| = subCircuitLength

Filtering algorithms:

subtour elimination : Caseau & Laburthe (ICLP'97)

allDifferent GAC algorithm: Régin (AAAI'94)

dominator-based filtering: Fages & Lorca (CP'11) (adaptive scheme by default, see implementation)

Parameters:

`vars - a vector of variables`

`offset - 0 by default but 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)`

`subCircuitLength - expected number of nodes in the circuit`

Returns:

`a subCircuit constraint`

subPath

```
default Constraint subPath(IntVar[] vars,
                           IntVar start,
                           IntVar end,
                           int offset,
                           IntVar SIZE)
```

Creates a subPath constraint which ensures that

the elements of vars define a path of SIZE vertices, leading from start to end

where vars[i] = offset+j means that j is the successor of i.

where vars[i] = offset+i means that vertex i is excluded from the path.

Moreover, vars[end-offset] = |vars|+offset

Requires : |vars|>0

Filtering algorithms: see subCircuit constraint

Parameters:

`vars - vector of variables which take their value in [offset,offset+|vars|]`

`start - variable indicating the index of the first variable in the path`

`end - variable indicating the index of the last variable in the path`

`offset - 0 by default but typically 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)`

`SIZE - variable indicating the number of variables to belong to the path`

Returns:

`a subPath constraint`

sum

```
default Constraint sum(IntVar[] vars,
                       String operator,
                       int sum)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum.

Parameters:

`vars - a collection of IntVar`

`operator - operator in {"=", "!=", ">","<",">=","<="}`

sum - an integer

Returns:

a sum constraint

sum

```
default Constraint sum(IntVar[] vars,
                       String operator,
                       int sum,
                       int minCardForDecomp)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum.

Parameters:

vars - a collection of IntVar

operator - operator in {"=", "!=", ">","<",">=","<="}

sum - an integer

minCardForDecomp - minimum number of cardinality threshold to a sum constraint to be decomposed

Returns:

a sum constraint

sum

```
default Constraint sum(IntVar[] vars,
                       String operator,
                       IntVar sum)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum.

Parameters:

vars - a collection of IntVar

operator - operator in {"=", "!=", ">","<",">=","<="}

sum - an IntVar

Returns:

a sum constraint

sum

```
default Constraint sum(IntVar[] vars,
```

```
                        String operator,
                        IntVar sum,
                        int minCardForDecomp)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum.

Parameters:

`vars - a collection of IntVar`

`operator - operator in {"=", "!=", ">","<",">=","<="}`

`sum - an IntVar`

`minCardForDecomp - minimum number of cardinality threshold to a sum constraint to be decomposed`

Returns:

`a sum constraint`

sum

```
default Constraint sum(BoolVar[] vars,
                        String operator,
                        int sum)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum. This constraint is much faster than the one over integer variables

Parameters:

`vars - a vector of boolean variables`

`sum - an integer`

sum

```
default Constraint sum(BoolVar[] vars,
                        String operator,
                        IntVar sum)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum. This constraint is much faster than the one over integer variables

Parameters:

`vars - a vector of boolean variables`

`sum - a variable`

sum

```
default Constraint sum(BoolVar[] vars,
                       String operator,
                       IntVar sum,
                       int minCardForDecomp)
```

Creates a sum constraint. Enforces that $\sum_{i \text{ in } |vars|} vars_i$ operator sum. This constraint is much faster than the one over integer variables

Parameters:

`vars - a vector of boolean variables`

`sum - a variable`

`minCardForDecomp - minimum number of cardinality threshold to a sum constraint to be decomposed`

table

```
default Constraint table(IntVar[] vars,
                         Tuples tuples)
```

Creates a table constraint specifying that the sequence of variables vars must belong to the list of tuples (or must NOT belong in case of infeasible tuples) Default configuration with GACSTR+ algorithm for feasible tuples and GAC3rm otherwise

Parameters:

`vars - variables forming the tuples`

`tuples - the relation between the variables (list of allowed/forbidden tuples)`

table

```
default Constraint table(IntVar[] vars,
                         Tuples tuples,
                         String algo)
```

Creates a table constraint, with the specified algorithm defined algo

- **CT+**: Compact-Table algorithm (AC),
- **GAC2001**: Arc Consistency version 2001 for tuples,
- **GAC2001+**: Arc Consistency version 2001 for allowed tuples,
- **GAC3rm**: Arc Consistency version AC3 rm for tuples,
- **GAC3rm+** (default): Arc Consistency version 3rm for allowed tuples,
- **GACSTR+**: Arc Consistency version STR for allowed tuples,
- **STR2+**: Arc Consistency version STR2 for allowed tuples,
- **FC**: Forward Checking.

- **MDD+**: uses a multi-valued decision diagram for allowed tuples (see mddc constraint),

Parameters:

`vars - variables forming the tuples`

`tuples - the relation between the variables (list of allowed/forbidden tuples). Should not be modified once passed to the constraint.`

`algo - to choose among {"TC+", "GAC3rm", "GAC2001", "GACSTR", "GAC2001+", "GAC3rm+", "FC", "STR2+"}`

tree

```
default Constraint tree(IntVar[] succs,
                        IntVar nbTrees)
```

Creates a tree constraint. Partition succs variables into nbTrees (anti) arborescences

succs[i] = j means that j is the successor of i.

and succs[i] = i means that i is a root

dominator-based filtering: Fages & Lorca (CP'11)

However, the filtering over nbTrees is quite light here

Parameters:

`succs - successors variables, taking their domain in [0,|succs|-1]`

`nbTrees - number of arborescences (=number of loops)`

Returns:

`a tree constraint`

tree

```
default Constraint tree(IntVar[] succs,
                        IntVar nbTrees,
                        int offset)
```

Creates a tree constraint. Partition succs variables into nbTrees (anti) arborescences

succs[i] = offset+j means that j is the successor of i.

and succs[i] = offset+i means that i is a root

dominator-based filtering: Fages & Lorca (CP'11)

However, the filtering over nbTrees is quite light here

Parameters:

`succs - successors variables, taking their domain in [offset,|succs|-1+offset]`

`nbTrees - number of arborescences (=number of loops)`

`offset - 0 by default but 1 if used within MiniZinc (which counts from 1 to n instead of from 0 to n-1)`

Returns:

`a tree constraint`

getDomainUnion

```
default int[] getDomainUnion(IntVar... vars)
```

Get the list of values in the domains of vars

Parameters:

`vars - an array of integer variables`

Returns:

`the list of values in the domains of vars`