

Reference manual



FICO® Xpress Optimization

Xpress Kalis

Reference Manual

Release 12.6

Last update 7 July, 2017

This material is the confidential, proprietary, and unpublished property of Fair Isaac Corporation. Receipt or possession of this material does not convey rights to divulge, reproduce, use, or allow others to use it without the specific written authorization of Fair Isaac Corporation and use must conform strictly to the license agreement.

The information in this document is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither Fair Isaac Corporation nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement.

©2005–2017 Fair Isaac Corporation and Artelys SA. All rights reserved. Permission to use this software and its documentation is governed by the software license agreement between the licensee and Fair Isaac Corporation (or its affiliate). Portions of the program may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall Fair Isaac Corporation or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this software and its documentation, even if Fair Isaac Corporation or its affiliates have been advised of the possibility of such damage. The rights and allocation of risk between the licensee and Fair Isaac Corporation (or its affiliates) are governed by the respective identified licenses in the software, documentation, or both.

Fair Isaac Corporation and its affiliates specifically disclaim any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The software and accompanying documentation, if any, provided hereunder is provided solely to users licensed under the Fair Isaac Software License Agreement. Fair Isaac Corporation and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under the Fair Isaac Software License Agreement.

FICO and Fair Isaac are trademarks or registered trademarks of Fair Isaac Corporation in the United States and may be trademarks or registered trademarks of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Xpress Kalis
Deliverable Version: A
Last Revised: 7 July, 2017
Version 12.6

How to Contact the Xpress Team

Information, Sales and Licensing

USA, CANADA AND ALL AMERICAS
Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Product Support

Email: Support@fico.com
(Please include 'Xpress' in the subject line)

Telephone:

NORTH AMERICA
Tel (toll free): +1 (877) 4FI-SUPP
Fax: +1 (402) 496-2224

EUROPE, MIDDLE EAST, AFRICA
Tel: +44 (0) 870-420-3777
UK (toll free): 0800-0152-153
South Africa (toll free): 0800-996-153
Fax: +44 (0) 870-420-3778

ASIA-PACIFIC, LATIN AMERICA, CARIBBEAN
Tel: +1 (415) 446-6185
Brazil (toll free): 0800-891-6146

For the latest news and Xpress software and documentation updates, please visit the Xpress website at <http://www.fico.com/xpress> or subscribe to our mailing list.

How to Contact Artelys

Artelys SA

Information and Sales: info-kalis@artelys.com
Licensing and Product Support: support-kalis@artelys.com

Tel: +33 1 44 77 89 00
Fax: +33 1 42 96 22 61

12, rue du Quatre Septembre
75002 Paris Cedex
France

For the latest news about Kalis, training course programs, and examples, please visit the Artelys website at <http://www.artelys.com>.

Contents

1	Introduction	1
1.1	Contents of this manual	1
1.2	Constraint programming overview	1
1.3	What is Xpress Kalis?	3
1.4	Prerequisites	3
I	Paradigm	4
2	Decision variables	5
3	Constraints	7
4	Enumeration and search strategy	9
5	Scheduling	14
5.1	Tasks	14
5.2	Resources	16
5.3	Schedule	17
6	Linear relaxations	21
6.1	Automatic relaxations	21
6.2	User defined relaxations	22
6.3	Usage of relaxations	23
6.4	Branching with relaxation	24
6.5	A simple hybrid example	24
II	Reference Manual	28
7	Constants	29
KALIS_FORWARD_CHECKING		31
KALIS_GEN_ARC_CONSISTENCY		32
KALIS_SLIM_UNREACHED		32
KALIS_SLIM_BY_NODES		32
KALIS_SLIM_BY_SOLUTIONS		33
KALIS_SLIM_BY_DEPTH		33
KALIS_SLIM_BY_TIME		33
KALIS_SLIM_BY_BACKTRACKS		33
KALIS_TLIM_UNREACHED		34
KALIS_TLIM_ABS_OPT		34
KALIS_TLIM_REL_OPT		34
KALIS_INPUT_ORDER		35
KALIS_MAX_DEGREE		35
KALIS_LARGEST_MIN		35
KALIS_LARGEST_MAX		36

KALIS_MAXREGRET_LB	36
KALIS_MAXREGRET_UB	36
KALIS_RANDOM_VARIABLE	37
KALIS_SDOMDEG_RATIO	37
KALIS_SMALLEST_DOMAIN	37
KALIS_SMALLEST_MAX	38
KALIS_SMALLEST_MIN	38
KALIS_WIDEST_DOMAIN	38
KALIS_MAX_TO_MIN	39
KALIS_MIN_TO_MAX	39
KALIS_MIDDLE_VALUE	39
KALIS_NEAREST_VALUE	39
KALIS_RANDOM_VALUE	40
KALIS_COPYRIGHT	40
KALIS_UNARY_RESOURCE	40
KALIS_DISCRETE_RESOURCE	40
KALIS_SMALLEST_ECT	41
KALIS_SMALLEST_EST	41
KALIS_SMALLEST_LCT	41
KALIS_SMALLEST_LST	42
KALIS_LARGEST_ECT	42
KALIS_LARGEST_EST	42
KALIS_LARGEST_LCT	42
KALIS_LARGEST_LST	43
KALIS_TASK_INPUT_ORDER	43
KALIS_TASK_RANDOM_ORDER	43
KALIS_DISJ_INPUT_ORDER	44
KALIS_DISJ_PRIORITY_ORDER	44
KALIS_TIMETABLING	44
KALIS_TASK_INTERVALS	45
KALIS_DISJUNCTIONS	45
KALIS_EDGE_FINDING	45
KALIS_INITIAL_SOLUTION	46
KALIS_OPTIMAL_SOLUTION	46
KALIS_RESET_PARAMS_ALL	46
KALIS_RESET_VAR_BOUNDS	47
KALIS_RESET_VAR_PRECISION	47
KALIS_RESET_OPT_PARAMS	48
KALIS_RESET_SEARCH_PARAMS	48
KALIS_TOPNODE_RELAX_SOLVER	48
KALIS_TREENODE_RELAX_SOLVER	49
KALIS_BILEVEL_RELAX_SOLVER	49
KALIS_LINEAR_CONSTRAINTS	49
KALIS_NON_LINEAR_CONSTRAINTS	49
KALIS_LOGICAL_CONSTRAINTS	50
KALIS_DISTANCE_CONSTRAINTS	50
KALIS_ALL_CONSTRAINTS	50
KALIS_MINIMIZE	51
KALIS_MAXIMIZE	51
KALIS_SOLVE_AS_LP	51
KALIS_SOLVE_AS_MIP	51
KALIS_RELAX_PRESOLVE	52
KALIS_RELAX_RCOSTS_PROPAG	52
KALIS_RELAX_RELOAD_BASIS	52
KALIS_RELAX_GLOBAL	52

KALIS_RELAX_ALGORITHM	53
KALIS_RELAX_OPT_TOL	53
KALIS_RELAX_MIP_REL_STOP	53
KALIS_RELAX_MIP_ABS_STOP	53
KALIS_PRIMAL_SIMPLEX	54
KALIS_DUAL_SIMPLEX	54
KALIS_BARRIER	54
KALIS_NETWORK_SIMPLEX	55
8 Parameters	56
KALIS_DEFAULT_LB	57
KALIS_DEFAULT_UB	57
KALIS_AUTO_PROPAGATE	57
KALIS_OPTIMIZE_WITH_RESTART	58
KALIS_NODES	58
KALIS_DEPTH	58
KALIS_SEARCH_LIMIT	58
KALIS_TOLERANCE_LIMIT	59
KALIS_BACKTRACKS	59
KALIS_COMPUTATION_TIME	59
KALIS_MAX_NODES	59
KALIS_MAX_SOLUTIONS	60
KALIS_MAX_DEPTH	60
KALIS_MAX_BACKTRACKS	60
KALIS_MAX_COMPUTATION_TIME	61
KALIS_MAX_NODES_BETWEEN_SOLUTIONS	61
KALIS_OPT_ABS_TOLERANCE	61
KALIS_OPT_REL_TOLERANCE	61
KALIS_CHECK_SOLUTION	62
KALIS_DEFAULT_CONTINUOUS_LB	62
KALIS_DEFAULT_CONTINUOUS_UB	62
KALIS_DEFAULT_PRECISION_VALUE	63
KALIS_DEFAULT_PRECISION_RELATIVITY	63
KALIS_DEFAULT_SCHEDULE_HORIZ_MIN	63
KALIS_DEFAULT_SCHEDULE_HORIZ_MAX	63
KALIS_DICHOTOMIC_OBJ_SEARCH	64
KALIS_USE_3B_CONSISTENCY	64
KALIS_VERBOSE_LEVEL	64
KALIS_AUTO_RELAX	64
KALIS_THREADS	65
KALIS_NARY_OBJ_SEARCH	65
KALIS_SCHEDULE_ENABLE_SHAVING	65
9 Subroutines	66
9.1 Constraints	66
cplinctr	67
and	69
or	70
equiv	71
implies	73
cpnlinctr	75
In	77
exp	78
distance	79
abs	81

all_different	83
cycle	85
maximum_minimum	90
occurrence	92
element	94
generic_binary_constraint	96
generic_nary_constraint	98
distribute	99
cumulative	101
disjunctive	103
producer_consumer	105
9.2 Constraint parameters	108
getarity	109
getpriority	110
setpriority	111
gettag	112
settag	113
setfirstbranch	114
getactivebranch	115
9.3 Variables	116
getlb	117
getub	118
getmiddle	119
getsize	120
getval	121
is_fixed	122
getdegree	123
gettarget	124
getrand	125
getnext	126
getprev	127
contains	128
is_equal	129
is_same	130
settarget	131
setdomain	132
setlb	133
setub	134
setval	135
setprecision	136
cp_show_var	137
9.4 Problem	138
cp_post	139
cp_propagate	140
cp_find_next_sol	141
cp_reset_search	142
cp_maximise	143
cp_maximize	144
cp_minimise	145
cp_minimize	146
getsol	147
cp_show_prob	148
cp_show_sol	149
cp_show_best_sol	150
getname	151

setname	152
cp_shave	153
cp_infeas_analysis	154
cp_save_state	155
cp_restore_state	156
path_order	157
cp_local_optimize	158
set_sol_as_target	159
cp_reset_params	160
set_reversible_attributes	161
setval	162
getval	163
setelt	164
getelt	165
getsize	166
9.5 Search	167
assign_var	168
assign_and_forbid	172
settle_disjunction	176
probe_assign_var	178
probe_settle_disjunction	179
split_domain	181
task_serialize	185
cp_set_branching	189
cp_show_stats	192
bs_group	193
group_serializer	194
gettag	195
9.6 Callbacks	196
cp_set_solution_callback	197
cp_set_node_callback	200
cp_set_branch_callback	203
9.7 Scheduling	206
is_fixed	208
set_task_attributes	209
setduration	210
addpredecessors	211
setpredecessors	212
addsuccessors	213
setsuccessors	214
cp_show_schedule	215
getstart	216
getend	217
getduration	218
getconsumption	219
getrequirement	221
getproduction	223
getprovision	225
is_consuming	227
is_requiring	228
is_producing	229
is_providing	230
cp_schedule	231
getmakespan	232
consumes	233

requires	236
produces	239
provides	242
set_resource_attributes	245
setcapacity	246
setmaxavailability	248
setminusage	250
getcapacity	252
setsetuptime	254
getsetuptime	255
is_fixed	256
cp_set_schedule_strategy	257
cp_get_default_schedule_strategy	258
resusage	259
getassignment	262
has_assignment	264
setidletimes	266
is_idletime	267
cp_close_schedule	268
9.8 Linear relaxations	269
KALIS_LARGEST_REDUCED_COST	270
KALIS_NEAREST_RELAXED_VALUE	271
cp_get_linrelax	272
get_linrelax_solver	273
fix_to_relaxed	274
set_verbose_level	275
cp_show_relax	276
cp_add_linrelax_solver	277
cp_remove_linrelax_solver	278
cp_clear_linrelax_solver	279
generate_cuts	280
lp_optimize	281
set_linrelax_solver_attribute	282
get_indicator	283
get_linrelax	284
export_prob	285
get_reduced_cost	286
get_relaxed_value	287
set_integer	288
Appendix	289
A Contacting FICO	289
Product support	289
Product education	289
Product documentation	289
Sales and maintenance	290
Related services	290
About FICO	290

Bibliography	291
Index	292

CHAPTER 1

Introduction

1.1 Contents of this manual

This reference manual documents version 12.6 of Xpress Kalis.

This document and the software described in this document are furnished under a license or non-disclosure agreement, and may be used or copied only within the terms of such a license or non-disclosure agreement.

1.2 Constraint programming overview

Constraint programming (CP) is a software technology becoming more widespread thanks to many successes with effective solving of large, real-life, particularly combinatorial, problems. It provides a powerful technique for different problems such as scheduling, timetabling, resource allocation, or network configuration.

Research results from different fields (operations research, artificial intelligence, discrete mathematics, graph theory) are all involved in the core of Constraint Programming packages. Constraint Programming allows for representing many problems in a way which is very close to a natural language description, thanks to semantic constraints. Benefits are important: fast prototyping, compact code, easy modification, and good performance. It allows for specifying powerful decision support systems.

In order to have a quick overview of this technique, here are the basic elements required for a problem description:

A *constraint satisfaction problem* (CSP) is defined by:

- A set of decision variables $V = \{x_1, \dots, x_n\}$
- For each variable, a set (or a range) of possible values called its *domain*
- A set of constraints on these variables.

The *arity* of a constraint corresponds to the number of variables that it involves. For discrete variables, domain values do not have to be consecutive (for example $\{1, 4, 6, 8\}$). For continuous variables, the domain is modelled as an interval.

An interesting notion is the *constraint graph*:

A CSP can be represented by a non-oriented graph where the edges symbolize the links between constraints and variables. The Figure 1.1 shows an example of a constraint graph for the following CSP:

Variables: x, y, z
Constraints: $x \neq y, y \neq z, z \neq x$

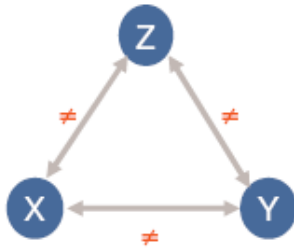


Figure 1.1: Constraint graph corresponding to the CSP

A *solution* to a CSP is an assignment of a value (belonging to its domain) to every variable, satisfying all the constraints.

In CP, constraints are used actively to deduce infeasible values and delete them from the domains of variables. This mechanism is called *constraint propagation*. It represents the core of Constraint Programming systems. Each constraint computes impossible values for its variables and informs other constraints. This process continues as long as new deductions are made.

Constraint propagation is associated with *tree search techniques* in order to find solutions or prove optimality. Each node and each decision will induce constraint propagation automatically. Many specific and efficient algorithms will be used in this propagation, but do not need to be known by the end-user. This allows persons who are familiar with problem modeling to quickly use such techniques for optimization or generation of solutions.

Solutions of a discrete CSP can be found by a systematic search on the set of possible assignments of values to variables. Other interval techniques are applied for continuous variables.

One may wish to find:

- just *one* feasible solution, without any choice preference
- *all* the solutions
- an *optimal* solution (or at least a nearly optimal solution) that optimizes a certain objective function defined on a subset of the variables of the problem

Solution search methods can be classified in two categories:

- Search methods that explore the space of the partial assignments
- Search methods that explore the space of complete assignments, generally in a stochastic way

The main reason for representing a problem as a CSP is that the formulation of the problem as a CSP is close to the original one in that: variables of the CSP directly correspond to the problem entities and the constraints can be expressed in a natural way without any translation to a set of linear inequalities as in the Mathematical Programming framework. This formulation is thus clearer, solutions are easier to represent, and search heuristics more direct.

1.3 What is Xpress Kalis?

Xpress Kalis is a Constraint Programming modeler based upon the Artelys Kalis solver. The software is provided in the form of a Mosel module, *kalis*. Xpress Mosel must be installed in order to be able to use Xpress Kalis. The graphical environment Xpress Workbench can be used to work with Xpress Kalis models. The module *kalis* extends the Mosel language with new ways of defining and solving Constraint Programming (CP) problems.

1.4 Prerequisites

This manual assumes some familiarity with the Mosel syntax. The reader is referred to the 'Mosel User Guide' for an introduction to working with Mosel. The 'Mosel Language Reference Manual' contains the complete documentation of the Mosel language. The 'Getting Started' manual explains how to work with Mosel models in Xpress Workbench.

I. Paradigm

CHAPTER 2

Decision variables

Decision variables are represented in Xpress Kalis by the types `cpvar` and `cpfloatvar`. They correspond respectively to finite domain variables taking their values in a given interval, or set of integers, called a domain and to continuous domain variables represented by real valued intervals. Conceptually the variables can be represented in the following way:

Decision variables are declared by using the standard Mosel syntax. For instance, the following code extract shows how to create a finite domain variable `my_cpvar`, a static array of `cpvar` `cpvar_array`, and a dynamic array of `cpfloatvar` `dyn_cpfloatvar_array`:

```
declarations
  my_cpvar          : cpvar
  cpvar_array       : array(1..10) of cpvar
  dyn_cpfloatvar_array : array(range) of cpfloatvar
end-declarations
```

Note that entries of dynamic arrays of domain variables (arrays declared as `dynamic` at their creation or arrays for which some or all of the index sets are not known at the declaration like `dyn_cpfloatvar_array` in the example above) must be created explicitly using the Mosel procedure `create`:

```
declarations
  dyn_cpfloatvar_array : array(range) of cpfloatvar
end-declarations
forall(i in 3..30) create(dyn_cpvar_array(i))
```

After its declaration, the second step in the creation of a domain variable consists of defining its domain with the procedure `setdomain`.

Domain variables are also used in the definition of constraints and search strategy.

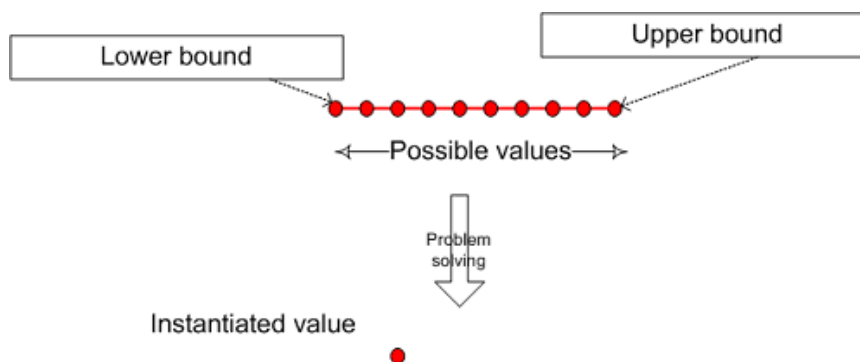


Figure 2.1: Conceptual representation of finite domain variables

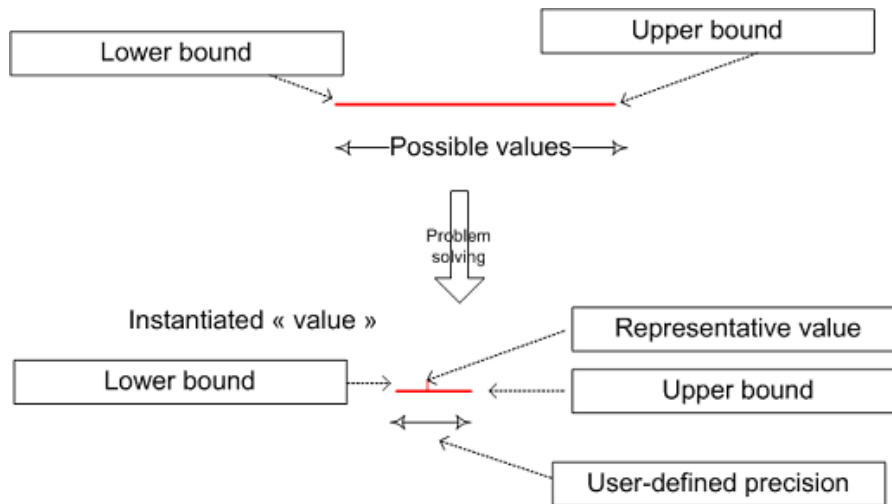


Figure 2.2: Conceptual representation of continuous variables

Xpress Kalis defines a set of functions for accessing and modifying `cpvar` and `cpfloatvar` states:

<code>contains</code>	Tests if a value is in the domain of a variable	p. 128
<code>cp_show_var</code>	Shows the current domain of the variable	p. 137
<code>getdegree</code>	Returns the degree of a variable	p. 123
<code>getlb</code>	Returns the current lower bound of a variable	p. 117
<code>getmiddle</code>	Returns the middle value of a variable	p. 119
<code>getnext</code>	Gets the next value in the domain of a finite domain variable	p. 126
<code>getprev</code>	Gets the previous value in the domain of a finite domain variable	p. 127
<code>getrand</code>	Returns a random value belonging to the domain of a variable	p. 125
<code>getsize</code>	Returns the cardinality of the variable domain	p. 120
<code>gettarget</code>	Returns the target value of a variable	p. 124
<code>getub</code>	Returns the current upper bound of a variable	p. 118
<code>getval</code>	Returns the instantiation value of a variable	p. 121
<code>is_equal</code>	Tests if two variable domains are equal	p. 129
<code>is_fixed</code>	Tests if the variable passed in argument is instantiated	p. 122
<code>is_same</code>	Tests if two decision variables represent the same variable	p. 130
<code>setdomain</code>	Sets the domain of a variable	p. 132
<code>setlb</code>	Sets the lower bound of a variable	p. 133
<code>setprecision</code>	Sets the precision relativity and value of a continuous variable	p. 136
<code>settarget</code>	Sets the target value of a variable	p. 131
<code>setub</code>	Sets the upper bound of a variable	p. 134
<code>setval</code>	Instantiate the value of a variable	p. 135

CHAPTER 3

Constraints

Constraints represent logical restrictions on the values that decision variables can simultaneously take and that must be satisfied. There are various types of constraints that can be stated within Xpress Kalis:

- Linear constraint (equation, inequality, disequality), e.g.: $x \neq y$, $x = y$, $x \leq y$ etc.
- Non-linear constraint (equation, inequality), e.g.: $x = y / z$, $x * y + 2 * \ln(z) = \exp(w)$, $x^2 + y^2 \leq z^2$ etc.
- Symbolic constraint, e.g.: *all-different*(x, y, z) meaning $(x \neq y) \wedge (x \neq z) \wedge (y \neq z)$
- Logical relation, e.g.: $x \neq y \Rightarrow z = 3$
- User-defined constraints such as *generic_binary_constraint*(x, y)

All constraints are represented by the type `cpctr`.

<code>abs</code>	Absolute value constraint	p. 81
<code>all_different</code>	All different constraint	p. 83
<code>and</code>	Conjunction composite constraint	p. 69
<code>cplnctr</code>	Linear constraints	p. 67
<code>cpnlnctr</code>	Non-linear constraints	p. 75
<code>cumulative</code>	Cumulative constraint	p. 101
<code>cycle</code>	Cycle constraint	p. 85
<code>disjunctive</code>	Disjunctive constraint	p. 103
<code>distance</code>	Distance constraint	p. 79
<code>distribute</code>	Distribute constraint with fixed bounds	p. 99
<code>element</code>	Element constraint	p. 94
<code>equiv</code>	Equivalence composite constraint	p. 71
<code>exp</code>	Exponential of a non-linear expression	p. 78
<code>generic_binary_constraint</code>	Generic Binary constraint	p. 96
<code>generic_nary_constraint</code>	Generic nary constraint	p. 98
<code>implies</code>	Implication composite constraint	p. 73

<code>ln</code>	Natural logarithm of a non-linear expression	p. 77
<code>maximum_minimum</code>	Maximum/minimum constraint	p. 90
<code>occurrence</code>	Occurrence constraint	p. 92
<code>or</code>	Disjunction composite constraint	p. 70
<code>producer_consumer</code>	Producer Consumer constraint	p. 105

CHAPTER 4

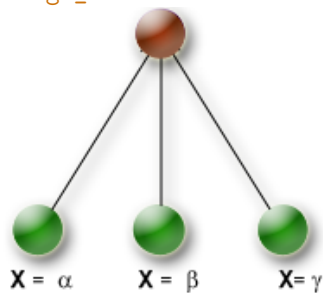
Enumeration and search strategy

Since the combination of constraint solving and propagation mechanism is usually not sufficient to reduce all variable domains to a single value, an enumeration needs to be added to the problem definition to find feasible solutions or to prove that no such solution exists.

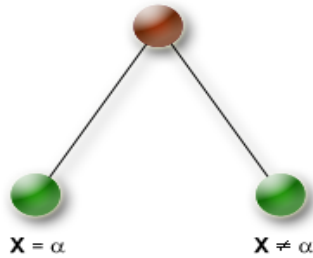
The search process is made by a branch and bound algorithm with depth-first exploration of the search tree. At each node, a propagation phase is triggered in order to detect possible inconsistencies and reduce the search space. If this phase detects an inconsistency, the algorithm backtracks and removes the effects of the previous decisions. If no inconsistency is detected, a branching process is applied recursively to the child nodes until a solution is found or until all the search space has been explored.

The way the branching is done is defined by branching schemes (type `cpbranching`). Six *branching schemes* are predefined in Xpress Kalis:

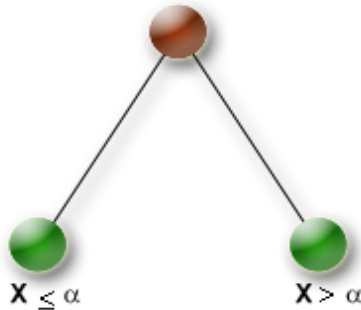
1. `assign_var`



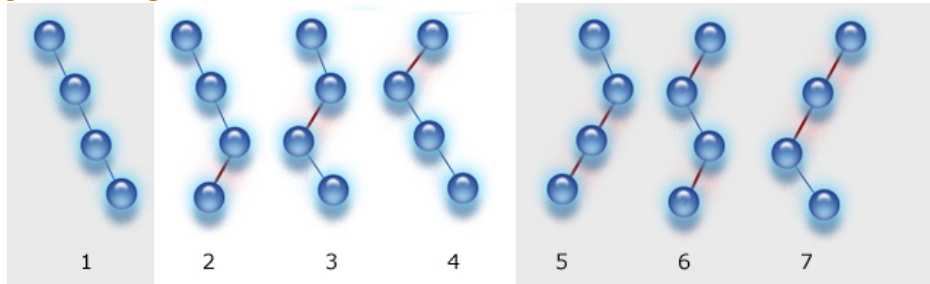
This branching scheme may be considered as the standard enumeration scheme in finite domain Constraint Programming. For a given branching variable it enumerates all values currently in its domain. A branch is formed by assigning a value to the branching variable, resulting in a variable number of branches per node (determined by the variable's domain size). Since this strategy tends to create a very wide search tree it is used preferably with small domains or if variable and value selection strategies are known that quickly lead to (good) feasible solutions.

2. `assign_and_forbid`

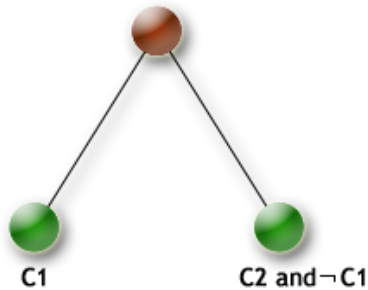
This branching scheme creates a binary search tree by creating two branches for a given branching variable. The first branch is formed by assigning a value to the branching variable, and the second branch, by forbidding this value for this variable.

3. `split_domain`

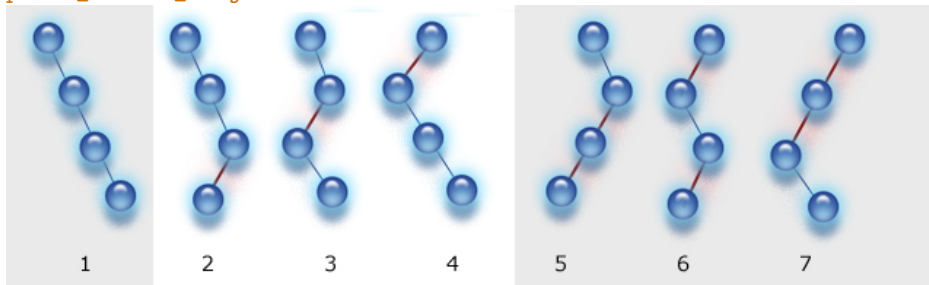
The `split_domain` strategy creates a binary search tree by splitting the domain of the branching variable into two intervals (all values less than or equal to the branching value in the first and all values greater than the branching value in the second). This strategy is the only possible choice for continuous variables and it is also applicable for finite domain variables. For the latter this strategy may be helpful if the initial domain sizes are relatively large. The strategy can be configured by choosing which branch to explore first and whether to stop applying it when domain sizes are reduced to less than a certain limit.

4. `probe_assign_var`

The `probe_assign_var` strategy is directly derived from the `assign_var` strategy which means that the branches created are the same. The difference lies in the completeness and the order in which branches are explored. Some of the possible branches are skipped by limiting the global number of times that the heuristic (first branch created) may fail and the branching process is stopped when this counter exceeds a specified value. The picture below shows the order in which the branches are explored when the domains of all variables are reduced to two values. This strategy can be useful to apply local search to an already known solution (combined with the `KALIS_NEAREST_VALUE` value selection heuristic) or to explore the neighborhood of a heuristic solution to find quickly a first solution and avoid the thrashing behavior of the chronological backtracking algorithm used during the resolution process.

5. `settle_disjunction`

The `settle_disjunction` strategy creates a binary search tree by branching on the two possibilities defined by a disjunction. Recall that a disjunction can be written as $c1 \text{ or } c2$ where $c1$ and $c2$ are two constraints, the `settle_disjunction` strategy will create a branch for each constraint stating that it must hold. The strategy can be configured by choosing the first branch of the disjunction that will be explored during the search process.

6. `probe_settle_disjunction`

The `probe_assign_var` strategy is directly derived from the `settle_disjunction` strategy which means that the branches created by this strategy are the same. The difference lies in the completeness and the order in which branches are explored. Some of the possible branches are skipped by limiting the total number of times that the heuristic (first branch created) may fail and the branching process is stopped when this counter exceeds a specified value. The picture below shows the order in which the branches are explored when the domains of all variables are reduced to two values. This strategy can be useful to apply local search to a disjunctive scheduling problem for which a solution is already known (combined with the `setfirstbranch` and `getactivebranch` methods).

These predefined branching schemes are used in conjunction with variable (resp. disjunction) and value selection heuristics that fully define the way how the search tree will be explored.

Xpress Kalis predefines several *variable selection heuristics*:

- `KALIS_SMALLEST_DOMAIN`
- `KALIS_MAX_DEGREE`
- `KALIS_SDOMDEG_RATIO`
- `KALIS_SMALLEST_MAX`
- `KALIS_SMALLEST_MIN`
- `KALIS_LARGEST_MAX`
- `KALIS_LARGEST_MIN`
- `KALIS_MAXREGRET_UB`

- `KALIS_MAXREGRET_LB`
- `KALIS_INPUT_ORDER`
- `KALIS_WIDEST_DOMAIN` (continuous variables)

... *value selection heuristics*:

- `KALIS_MAX_TO_MIN`
- `KALIS_MIN_TO_MAX`
- `KALIS_MIDDLE_VALUE`
- `KALIS_RANDOM_VALUE`

... *and disjunction selection heuristics*:

- `KALIS_DISJ_INPUT_ORDER`
- `KALIS_DISJ_PRIORITY_ORDER`

Xpress Kalis offers several possibilities to define the search strategy:

- Use the predefined strategy provided by Xpress Kalis.
- Build custom search strategies based on combinations of predefined branching schemes, variable and value selection heuristics.
- Define custom heuristics for variable and value selection.
- Add an optimization function (cost or objective function) for an enumeration and thus search for a feasible solution with the best (minimum or maximum) cost (optimal solution). We refer to this case as *optimization*, although, in practical applications one is typically only interested in good solutions that are found quickly, without necessarily spending time in proving that, indeed, the best solution is found.

The following example shows a specific problem and branching strategy with the corresponding search tree:

```
declarations
  x      : cpvar
  y      : cpvar
  strategy : array(1..2) of cpbranching
end-declarations
1 <= x; x <= 3
1 <= y; y <= 3
strategy(1) := split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX, {x})
strategy(2) := assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX, {y})
cp_set_branching(strategy)
```

<code>assign_and_forbid</code>	assign_and_forbid branching scheme	p. 172
<code>assign_var</code>	assign_var branching scheme	p. 168
<code>bs_group</code>	Create a group of branching schemes	p. 193
<code>cp_set_branching</code>	Sets the strategy to use during the search for a solution	p. 189

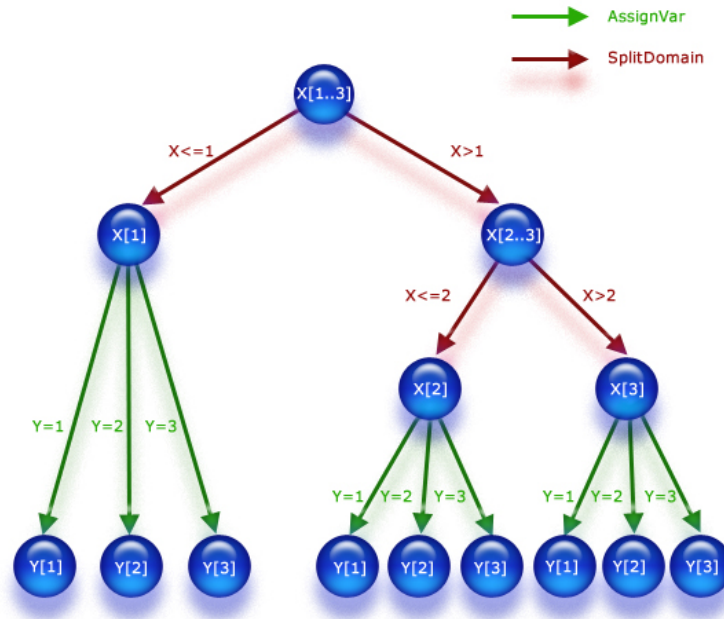


Figure 4.1: Example of search tree

<code>cp_show_stats</code>	Shows some statistics about the search	p. 192
<code>gettag</code>	Gets the tag of a constraint	p. 112
<code>group_serializer</code>	Creates a branching scheme Group Serializer	p. 194
<code>probe_assign_var</code>	probe_assign_var branching scheme	p. 178
<code>probe_settle_disjunction</code>	probe_settle_disjunction branching scheme	p. 179
<code>settle_disjunction</code>	settle_disjunction branching scheme	p. 176
<code>split_domain</code>	split_domain branching scheme	p. 181
<code>task_serialize</code>	task_serialize branching scheme	p. 185

CHAPTER 5

Scheduling

Scheduling plays an important role in manufacturing and engineering where it can have a major impact on the productivity of a process. The goal of scheduling is to optimize the production process by telling which operation to do, when, with which staff, and using which equipment.

Let's take the example of constructing a house. This task can be broken down into a set of smaller tasks:

- Grading and preparation of the site
- Foundations
- Framing
- Installation of windows and doors
- Roofing
- Siding
- Electricity
- Plumbing
- Underlayment
- Painting
- etc...

All these tasks must be carried out in a certain order: for example, laying the foundations cannot be done before the excavation and grading of the site. Moreover, certain tasks require specific resources (you need a plumber to do plumbing, a concrete mixer to do the carpentry etc.).

Tasks and resources are represented in Xpress Kalis by special purpose types such as `cpresource` and `cptask`.

5.1 Tasks

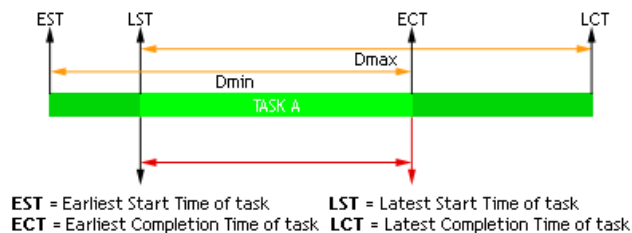
A `cptask` is represented by three decision variables of type `cpvar`:

- *start* representing the starting time of the task
- *end* representing the ending time of the task
- *duration* representing the duration of the task

These three structural variables are linked by Xpress Kalis with the following constraint:
 $start + duration = end$.

- The starting time variable represents two specific parameters of the task:
 - the *Earliest Start Time* (EST, represented by the lower bound of the 'start' variable) and its *Latest Start Time* (LST, represented by the upper bound of the 'start' variable).
- The end variable represents two specific parameters of the task:
 - the *Earliest Completion Time* (ECT, represented by the lower bound of the 'end' variable) and its *Latest Completion Time* (LCT, represented by the upper bound of the 'end' variable).
- The duration variable represents two specific parameters of the task:
 - the *minimum task duration* (Dmin, represented by the lower bound of the 'duration' variable) and the *maximum task duration* (Dmax, represented by the upper bound of the 'duration' variable).

The picture below illustrates this:



retrieved with the following methods:

Task attributes can be assigned or

- `getstart`
- `getend`
- `getduration`
- `setduration`
- `getname`
- `setname`
- `is_fixed`
- `addpredecessors`
- `setpredecessors`
- `addsuccessors`
- `setsuccessors`
- `setsetuptime`

5.2 Resources

A resource can be of two different types:

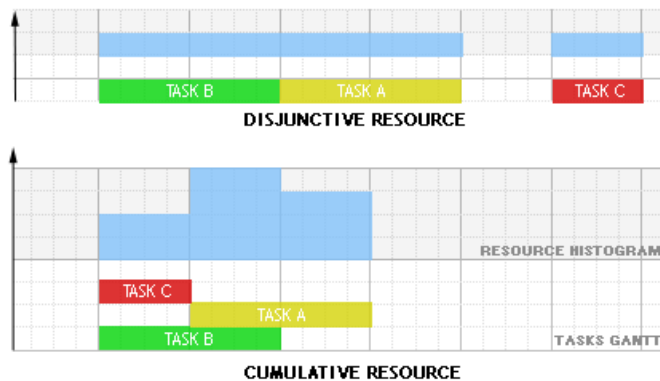
- *Disjunctive* when the resource can process only one task at a time (KALIS_UNARY_RESOURCE).
- *Cumulative* when the resource can process several tasks at the same time (KALIS_DISCRETE_RESOURCE).

Traditional examples of disjunctive resources are Jobshop scheduling problems. Cumulative resources are used for the Resource-Constrained Project Scheduling Problem (RCPSP)

Note that a disjunctive resource is semantically equivalent to a cumulative resource with maximal capacity one and unit resource usage for each task using this resource but this equivalence does not hold in terms of constraint propagation.

When defining a resource with Xpress Kalis its initial capacity is specified. For a disjunctive resource the capacity value is always equal to one. The structural capacity of a resource does not vary over time but a maximal temporal capacity can be imposed at any point of time with the procedure `setcapacity`.

The following graphic shows an example with three tasks A, B and C executing on a disjunctive resource and on a cumulative resource with resource usage 3 for task A, 1 for task B, and 1 for task C.



Tasks require, provide, consume and produce resources:

- A task **requires** a resource if some amount of the resource capacity must be made available for the execution of the activity. The capacity is renewable, this means the required capacity becomes available again after the end of the task.
- A task **provides** a resource if some amount of the resource capacity is made available through the execution of the task. The capacity is renewable, that is, the provided capacity is available only during the execution of the task.
- A task **consumes** a resource if some amount of the resource capacity must be made available for the execution of the task and the capacity is non-renewable, that is, the consumed capacity is no longer available at the end of the task.
- A task **produces** a resource if some amount of the resource capacity is made available through the execution of the task and the capacity is non-renewable, that is, the produced capacity is definitively available after the start of the task.

Task productions, requirements, provisions and productions can be accessed with the following methods:

- `consumes`
- `requires`
- `provides`
- `produces`
- `getconsumption`
- `getrequirement`
- `getprovision`
- `getproduction`
- `is_consuming`
- `is_requiring`
- `is_providing`
- `is_producing`

Resources and tasks are declared by using the standard Mosel syntax. For instance, the following code extract shows how to create a task `my_cptask`, a static array of resources `cpresource_array`, and a dynamic array of cptask `dyn_cptask_array`:

```
declarations
  my_cptask      : cptask
  cresource_array: array(1..10) of cresource
  dyn_cptask_array: array(range) of cptask
end-declarations
```

Note that entries of dynamic arrays of tasks or resources (arrays declared as `dynamic` at their creation or arrays for which some or all of the index sets are not known at the declaration like `dyn_cptask_array` in the example above) must be created explicitly using the Mosel procedure `create`:

```
declarations
  dyn_cptask_array: array(range) of cptask
end-declarations
forall(i in 3..30) create (dyn_cptask_array(i))
```

After its declaration, the second step in the creation of a task or resource consists of defining some attributes with the procedures `set_task_attributes` and `set_resource_attributes`.

5.3 Schedule

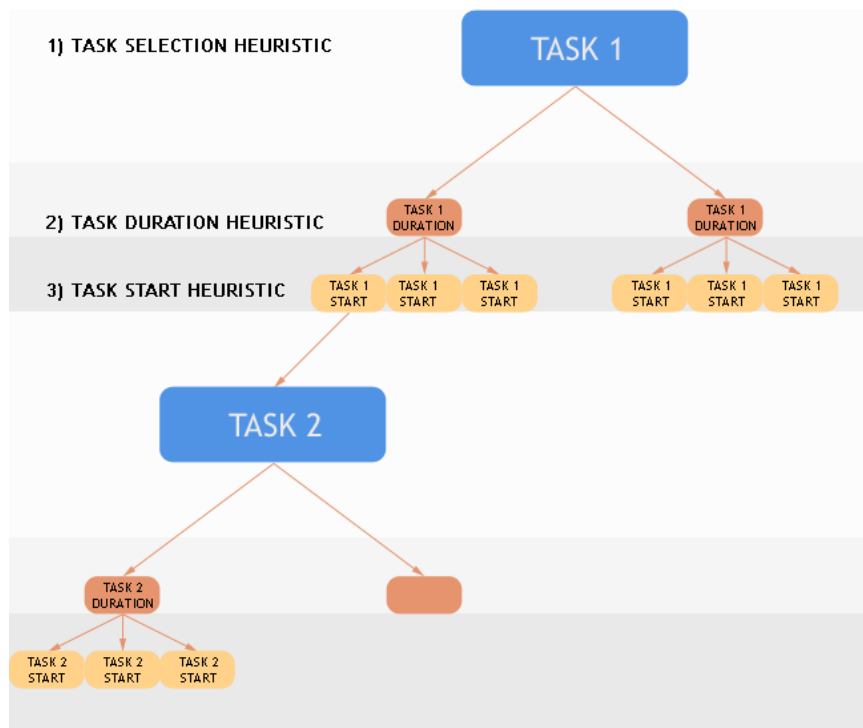
Tasks and resources form a *schedule*. The `cp_schedule` method allows to optimize a schedule with respect to any objective variable and implements advanced scheduling techniques specialized in makespan minimization. The creation of the schedule makespan can be automated by calling the `getmakespan` method that ensures automatically its computation.

A custom scheduling optimization strategy can be specified by using the `task_serialize` branching scheme to select the task to be scheduled and value choice heuristics for its start and duration variables.

The task selection method can be any user Mosel method or it may be configured with the predefined strategies of Xpress Kalis:

- `KALIS_SMALLEST_ECT`
- `KALIS_SMALLEST_EST`
- `KALIS_SMALLEST_LCT`
- `KALIS_SMALLEST_LST`
- `KALIS_LARGEST_ECT`
- `KALIS_LARGEST_EST`
- `KALIS_LARGEST_LCT`
- `KALIS_LARGEST_LST`

The picture below illustrates the definition of a task-based branching strategy:



Xpress Kalis defines a set of functions for accessing and modifying `cptask` and `cpresource` states:

<code>addpredecessors</code>	Adds a set of predecessors for a task	p. 211
<code>addsuccessors</code>	Adds a set of tasks as successors of a task	p. 213
<code>consumes</code>	Sets the minimal and maximal amount of resource consumed by a task for a particular resource	p. 233

<code>cp_close_schedule</code>	Close the schedule.	p. 268
<code>cp_get_default_schedule_strategy</code>	Gets the default schedule search strategy of <code>cp_schedule</code>	p. 258
<code>cp_schedule</code>	Optimizes the schedule with respect to an objective variable.	p. 231
<code>cp_set_schedule_strategy</code>	Sets the schedule search strategy for <code>cp_schedule</code>	p. 257
<code>cp_show_schedule</code>	Shows a textual representation of the current schedule	p. 215
<code>getassignment</code>	Gets the cpvar representing the assignment of a task for a particular resource.	p. 262
<code>getcapacity</code>	Get the maximal capacity of a resource for a specific time period.	p. 252
<code>getconsumption</code>	Gets the cpvar representing the consumption of a task for a particular resource	p. 219
<code>getduration</code>	Gets the cpvar representing a task duration	p. 218
<code>getend</code>	Gets the cpvar representing a task completion time	p. 217
<code>getmakespan</code>	Gets the cpvar representing the makespan of the schedule.	p. 232
<code>getproduction</code>	Gets the cpvar representing the production of a task for a particular resource	p. 223
<code>getprovision</code>	Gets the cpvar representing the provision of a task for a particular resource	p. 225
<code>getrequirement</code>	Gets the cpvar representing the requirement of a task for a particular resource	p. 221
<code>getsetuptime</code>	Gets the sequence dependent setup times between two tasks	p. 255
<code>getstart</code>	Gets the cpvar representing a task start time	p. 216
<code>has_assignment</code>	Tests whether an assignment decision variable for a task and a particular resource exists.	p. 264
<code>is_consuming</code>	Tests whether a task consumes a specific resource	p. 227
<code>is_fixed</code>	Tests if the variable passed in argument is instantiated	p. 122
<code>is_idletime</code>	Tests if a timestep is an idle timestep for a resource.	p. 267
<code>is_producing</code>	Tests whether a task produces a specific resource	p. 229
<code>is_providing</code>	Tests whether a task provides a specific resource	p. 230
<code>is_requiring</code>	Tests whether a task requires a specific resource	p. 228
<code>produces</code>	Sets the minimal and maximal amount of resource produced by a task for a particular resource	p. 239
<code>provides</code>	Sets the minimal and maximal amount of resource provided by a task for a particular resource.	p. 242
<code>requires</code>	Sets the minimal and maximal amount of resource required by a task for a particular resource	p. 236
<code>resusage</code>	Creates a resource usage	p. 259

<code>set_resource_attributes</code>	Sets some attributes for a resource	p. 245
<code>set_task_attributes</code>	Sets some attributes for a task	p. 209
<code>setcapacity</code>	Sets the maximal capacity of a resource between two time bounds. p. 246	
<code>setduration</code>	Sets the duration of a task	p. 210
<code>setidletimes</code>	Specifies the set of timesteps where a resource is idle.	p. 266
<code>setmaxavailability</code>	Sets the maximal capacity of a resource between two time bounds. p. 248	
<code>setminusage</code>	Sets the minimum usage of a resource between two time bounds. p. 250	
<code>setpredecessors</code>	Sets the tasks that must precede a task	p. 212
<code>setsetuptime</code>	Sets sequence dependent setup times between two tasks	p. 254
<code>setsuccessors</code>	Sets the set of tasks that must succeed a task	p. 214

CHAPTER 6

Linear relaxations

Some parts of an optimization problem may be best expressed and solved in the declarative (solver-independent) manner of Mathematical Programming (MP). Other parts benefit from constraint propagation and search directions provided by a CP solver. So why not trying to get the best of both worlds by combining the MP and CP approaches?

Xpress Kalis allows you to benefit from CP and MIP hybridization in a clean and easy way by means of linear relaxations of the CP model. Xpress Kalis can build automatically several linear (or mixed integer linear) relaxations of the CP problem and use Xpress Optimizer to solve them.

This process can be fully configured ranging from a complete user definition of the relaxation and choice of when to launch the relaxation to a fully automatic relaxation.

Xpress Kalis uses a double modeling approach by exchanging automatically and in bidirectional way, informations such as objective bounds, infeasibility, optimal relaxed solutions and reduced costs between the cp solver and the linear relaxations solver(s) during the search for a solution.

Note: the linear relaxation functionality of Xpress Kalis can only be used if Xpress Optimizer is installed and licensed.

6.1 Automatic relaxations

The following constraints can be relaxed automatically with Xpress Kalis :

- Linear constraints
- all-different
- occurrence
- distribute
- Min/Max
- absolute value
- distance
- element
- cycle
- logical (implies, or ,and ,equiv)

To obtain an automatic relaxation of the whole problem, Xpress Kalis takes the intersection of the relaxations of the constraints of the constraint programming model. Such a relaxation is obtained by a simple call to the function `cp_get_linrelax`

Note that this method is parameterized with an integer parameter since Xpress Kalis offers several relaxations for each constraint. In the present case, 0 means an LP oriented relaxation and 1 means a MIP oriented relaxation.

6.2 User defined relaxations

Xpress Kalis allows the user to define his own relaxation for the problem and provides several operators, functions and methods for this purpose:

- Overloaded operators for the specification of linear inequalities in a linear relaxation.
- The method `get_linrelax(ctr: cpctr)` returns the automatic relaxation for a constraint.
- The `set_integer` method turns variables in the relaxation into discrete variables (by default all relaxation variables are considered as continuous).
- Variables can be defined either in both, CP model and relaxation, or only in the CP model, or only in the relaxation (auxiliary variables).

Auxiliary variables (of the type `cpauxvar`) are additional variables that are used only in the definition of the relaxation and not in the formulation of the CP problem. For example, they can be used to linearize non-linear constraints or to express choices in the relaxation.

The CP model will not see these variables (no propagation and no branching) except for one particular kind of auxiliary variables called *indicators*. These indicator variables are 0-1 integer variables that are linked to `cpvar` variables. There is one indicator variable per value in the domain of a `cpvar`, and the indicator takes the value 1 if and only if this `cpvar` is instantiated to the value associated with the indicator variable. Bounds on indicator variables are automatically propagated by Xpress Kalis and reduced costs provided by the relaxation are retro-propagated to the CP variables by the means of reduced costs fixing (a kind of propagation reasoning on optimality).

Indicator variables are created and retrieved by a call to the `get_indicator` function.

The following examples show some expressions that can be used in the definition of custom relaxations:

```

declarations
myrelax: cpllinrelax           ! A linear relaxation
a1,a2 : cpauxvar               ! Auxiliary variables for relaxation
taux  : array(1..4) of cpauxvar ! Array of auxiliary variables

x1,x2,x3: cpvar                ! Finite domain variables
z       : cpfloatvar           ! CP variable with continuous domain
CAlld   : cpctr                ! A CP constraint
end-declarations

! Define an 'alldifferent' constraint
CAlld := all_different({x1,x2,x3})

! Build an automatic 'LP' oriented linear relaxation
myrelax1 := cp_get_linrelax(0)

! Setting bounds on the auxiliary variables
setdomain(a2,0,1)
setdomain(a1,0.56,18.67)

```

```
! Adding linear inequalities to the relaxation
myrelax += 3*a1      <= 3
myrelax += a1+a2     >= 3.1
myrelax += 2*a1-4*a2 = 3
myrelax += a1-x1     <= 3.4
myrelax += a1+4*a2-z <= 3
myrelax += get_linrelax(CA1ld,0)
myrelax +=
    get_indicator(x1,1) + get_indicator(x2,1) + get_indicator(x3,1) <= 1
myrelax += sum(i in 1..4) taux(i) = 4

! Set integrality condition for variables in the relaxation
set_integer(myrelax,a2,true)
set_integer(myrelax,x1,true)

! Output the resulting relaxation to the screen
cp_show_relax(myrelax)
```

6.3 Usage of relaxations

Xpress Kalis can create several distinct relaxations of a CP problem and solve them.

The optimal solution of these relaxations provides bounds for the CP branch-and-bound process and prunes the search tree with reduced costs propagation (optimality reasoning).

To solve a linear relaxation, Xpress Kalis uses relaxation solvers (of the type `cplinrelaxsolver`) that define for the relaxation:

- An objective variable for the relaxation
- An optimization sense (either `KALIS_MINIMIZE` or `KALIS_MAXIMIZE`)
- A configuration defining when and what to do with the relaxation.
- A flag indicating whether to solve the relaxation as a pure LP problem or as a MIP.(either `KALIS_SOLVE_AS_LP` or `KALIS_SOLVE_AS_MIP`)

Several predefined configurations are implemented within Xpress Kalis:

- `KALIS_TOPNODE_RELAX_SOLVER`: This configuration solves the relaxation only at the top node and provides bounds for the objective variable.
- `KALIS_TREENODE_RELAX_SOLVER`: This configuration solves the relaxation at each node of the search tree. It provides bounds for the objective variable and performs reduced costs propagation.
- `KALIS_BILEVEL_RELAX_SOLVER`: This configuration solves the relaxation whenever all the variables (by default all the discrete variables such as `cpvar`) of a user defined set are instantiated. After solving of the relaxation, all the other variables are instantiated to their optimal value in the relaxation. The main use of the bi-level configuration is to decompose the CP model simply and automatically into a master problem and a slave problem.

The user can also take full control over the relaxation solver by the means of callbacks.

A relaxation solver is obtained by specifying three callbacks functions:

- `must_relax`: This function with no argument and returning a Boolean must return true whenever the users wants to solve the relaxation

- `before_relax`: This procedure with no argument is called just before the relaxation is performed
- `after_relax(status: integer)`: This procedure with one argument is called after the relaxation has been solved. The status parameter indicates whether the relaxation is optimal (0) or infeasible (1). This callback can be used, for example, to instantiate some variables to their values in the optimal solution of the relaxation.

In addition to the user-configurable relaxations, Xpress Kalis provides a fully automated mode for using linear relaxations, enabled by the control parameter `auto_relax`.

6.4 Branching with relaxation

Xpress Kalis defines predefined branching schemes and value and variable selection heuristics based on the optimal solution of a relaxation:

- `KALIS_LARGEST_REDUCED_COST` variable selection heuristic selects the variable with the largest reduced cost in the optimal solution of the relaxation used by the specified relaxation solver.
- `KALIS_NEAREST_RELAXED_VALUE` value selection heuristic selects the value in the domain of the variable that is the nearest (in L1) to the value of this variable in the optimal solution of the relaxation used by the specified relaxation solver.

Moreover, the `fix_to_relaxed` method can be called during the tree search process to instantiate all the continuous variables to their values in the optimal solution of a relaxation.

This can be useful with some specific problem structures and in combination with the `KALIS_BILEVEL_RELAX_SOLVER` configuration for the relaxation solver to obtain an automatic logical benders decomposition of the solving process.

6.5 A simple hybrid example

Consider the following knapsack problem with an additional non linear constraint: all-different

$$\begin{aligned}
 &\text{minimize} && 5 \cdot x_1 + 8 \cdot x_2 + 4 \cdot x_3 \\
 &\text{s. t.} && 3 \cdot x_1 + 5 \cdot x_2 + 2 \cdot x_3 \geq 30 \\
 &&& \text{all-different}(x_1, x_2, x_3) \\
 &&& x_j \in \{1, 3, 8, 12\} \text{ for } j = 1, 2, 3
 \end{aligned}$$

A pure and straightforward CP approach for formulating and solving this problem is shown below:

```

model "Knapsack with side constraints"
uses "kalis"

declarations
  x1,x2,x3: cpvar          ! Decision variables
  benefit : cpvar          ! The objective to minimize
end-declarations

! Enable output printing
setparam("kalis_verbose_level", 1)

! Setting name of variables for pretty printing

```

```
setname(x1,"x1"); setname(x2,"x2"); setname(x3,"x3")
setname(benefit,"benefit")

! Set initial domains for variables
setdomain(x1, {1,3,8,12})
setdomain(x2, {1,3,8,12})
setdomain(x3, {1,3,8,12})

! Knapsack constraint
3*x1 + 5*x2 + 2*x3 >= 30

! Additional global constraint
all_different({x1,x2,x3})

! Objective function
benefit = 5*x1 + 8*x2 + 4*x3

! Initial propagation
res := cp_propagate

! Display bounds on objective after constraint propagation
writeln("Constraints propagation objective ", benefit)

! Solve the problem
if (cp_minimize(benefit)) then
  cp_show_sol                      ! Output optimal solution to screen
end-if

end-model
```

This model formulation can be augmented by the definition of a linear relaxation.

We start by getting an automatic relaxation of the problem by a call to the function `cp_get_linrelax`. The resulting relaxation can be displayed (printed to the standard output) with a call to the procedure `cp_show_relax`.

From the linear relaxation a linear relaxation *solver* is built with a call to the `get_linrelax_solver` method. Note that the `KALIS_TOPNODE_RELAX_SOLVER` argument passed to the method indicates that we just want to solve the linear relaxation at the top node of the CP search tree.

Having obtained the linear relaxation solver, we need to add it to the search process by a call to `cp_add_linrelax_solver`. Of course, Xpress Kalis is not limited to one relaxation so several solvers can be defined and added to the search process.

The model definition is completed by specifying a 'MIP style' branching scheme that branches first on the variables with largest reduced cost and tests first the values nearest to the optimal solution of the relaxation. The invocation of the search and solution display remain the same as in the CP model.

This is the full hybrid model:

```
model "Knapsack with side constraints"
uses "kalis"

declarations
  x1,x2,x3: cpvar                      ! Decision variables
  benefit : cpvar                      ! The objective to minimize
end-declarations

! Enable output printing
setparam("kalis_verbose_level", 1)

! Setting name of variables for pretty printing
setname(x1,"x1"); setname(x2,"x2"); setname(x3,"x3")
setname(benefit,"benefit")

! Set initial domains for variables
```

```
setdomain(x1, {1,3,8,12})
setdomain(x2, {1,3,8,12})
setdomain(x3, {1,3,8,12})

! Knapsack constraint
3*x1 + 5*x2 + 2*x3 >= 30

! Additional global constraint
all_different({x1,x2,x3})

! Objective function
benefit = 5*x1 + 8*x2 + 4*x3

! Initial propagation
res := cp_propagate

! Display bounds on objective after constraint propagation
writeln("Constraints propagation objective ", benefit)

declarations
myrelaxall: cplnrelax
end-declarations

! Build an automatic 'LP' oriented linear relaxation
myrelaxall:= cp_get_linrelax(0)

! Output the relaxation to the screen
cp_show_relax(myrelaxall)

mysolver:= get_linrelax_solver(myrelaxall, benefit, KALIS_MINIMIZE,
                             KALIS_SOLVE_AS_MIP, KALIS_TOPNODE_RELAX_SOLVER)

! Define the linear relaxation
cp_add_linrelax_solver(mysolver)

! Define a 'MIP' style branching scheme using the solution of the
! optimal relaxation
cp_set_branching(assign_var(KALIS_LARGEST_REDUCED_COST(mysolver),
                           KALIS_NEAREST_RELAXED_VALUE(mysolver)))

! Solve the problem
if (cp_minimize(benefit)) then
  cp_show_sol                      ! Output optimal solution to screen
end-if

end-model
```

You will find below the list of relaxation related functions and procedures defined within Xpress Kalis:

<code>cp_add_linrelax_solver</code>	Add a linear relaxation solver to the linear relaxation solver list	p. 277
<code>cp_clear_linrelax_solver</code>	Clear the linear relaxation solver list	p. 279
<code>cp_get_linrelax</code>	Returns an automatic relaxation of the cp problem	p. 272
<code>cp_remove_linrelax_solver</code>	Remove a linear relaxation solver from the linear relaxation solver list	p. 278
<code>cp_show_relax</code>	Pretty printing of a linear relaxation	p. 276
<code>export_prob</code>	Export the linear relaxation in LP format	p. 285
<code>fix_to_relaxed</code>	Fix the continuous variables to their optimal value in the relaxation solver passed in argument	p. 274

<code>generate_cuts</code>	Generate and add cuts to the relaxation passed in parameters	p. 280
<code>get_indicator</code>	Get an indicator variable for a given variable and a value.	p. 283
<code>get_linrelax</code>	Get the linear relaxation for a constraint	p. 284
<code>get_linrelax_solver</code>	Returns a linear relaxation solver from a linear relaxation, an objective variables and some configuration parameters	p. 273
<code>get_reduced_cost</code>	Get a reduced cost value from a linear relaxation solver	p. 286
<code>get_relaxed_value</code>	Returns the optimal relaxed value for a variable in a relaxation	p. 287
<code>KALIS_LARGEST_REDUCED_COST</code>	Get a largest reduced cost variable selector from a linear relaxation solver	p. 270
<code>KALIS_NEAREST_RELAXED_VALUE</code>	Get a nearest relaxed value selector from a linear relaxation solver	p. 271
<code>lp_optimize</code>	Launch LP/MIP solver without CP branching.	p. 281
<code>set_integer</code>	Set integrality flag for a variable in a linear relaxation	p. 288
<code>set_linrelax_solver_attribute</code>	Parameter setting for a linear relaxation solver.	p. 282
<code>set_verbose_level</code>	Set the verbose level for a specific linear relaxation solver	p. 275

II. Reference Manual

CHAPTER 7

Constants

KALIS_ALL_CONSTRAINTS	Set of all the constraints	p. 50
KALIS_BARRIER	Set the solution algorithm for the relaxation to barrier	p. 54
KALIS_BILEVEL_RELAX_SOLVER	Bilevel node relaxation solver configuration	p. 49
KALIS_COPYRIGHT	Xpress Kalis copyright information	p. 40
KALIS_DISCRETE_RESOURCE	Discrete resource	p. 40
KALIS_DISJ_INPUT_ORDER	Input order disjunction selection heuristic	p. 44
KALIS_DISJ_PRIORITY_ORDER	Input order disjunction selection heuristic	p. 44
KALIS_DISJUNCTIONS	Disjunctions propagation type for unary resource constraints	p. 45
KALIS_DISTANCE_CONSTRAINTS	Set of the distance constraints	p. 50
KALIS_DUAL_SIMPLEX	Set the solution algorithm for the relaxation to dual simplex	p. 54
KALIS_EDGE_FINDING	Tasks Interval + Edge finding propagation type for discrete resource constraints	p. 45
KALIS_FORWARD_CHECKING	Forward checking propagation type for the all_different constraint	p. 31
KALIS_GEN_ARC_CONSISTENCY	Generalized arc consistency propagation type for the all_different constraint	p. 32
KALIS_INITIAL_SOLUTION	Schedule search strategy choice: heuristic solution	p. 46
KALIS_INPUT_ORDER	Input Order variable selection heuristic	p. 35
KALIS_LARGEST_ECT	Largest Earliest Completion time task selection heuristic	p. 42
KALIS_LARGEST_EST	Largest Earliest Start time task selection heuristic	p. 42
KALIS_LARGEST_LCT	Largest Latest Completion time task selection heuristic	p. 42
KALIS_LARGEST_LST	Largest Latest Start time task selection heuristic	p. 43
KALIS_LARGEST_MAX	Largest maximum variable selection heuristic	p. 36
KALIS_LARGEST_MIN	Largest minimum variable selection heuristic	p. 35
KALIS_LINEAR_CONSTRAINTS	Set of linear constraints	p. 49
KALIS_LOGICAL_CONSTRAINTS	Set of logical constraints	p. 50
KALIS_MAX_DEGREE	Maximum constraint graph degree variable selection heuristic	p. 35

KALIS_MAX_TO_MIN	Maximum to minimum value selection heuristic	p. 39
KALIS_MAXIMIZE	Maximize objective in linear relaxation	p. 51
KALIS_MAXREGRET_LB	Maximum regret on the lower bound variable selection heuristic	p. 36
KALIS_MAXREGRET_UB	Maximum regret on the upper bound variable selection heuristic	p. 36
KALIS_MIDDLE_VALUE	Middle value selection heuristic	p. 39
KALIS_MIN_TO_MAX	Minimum to maximum value selection heuristic	p. 39
KALIS_MINIMIZE	Minimize objective in linear relaxation	p. 51
KALIS_NEAREST_VALUE	Nearest to target value selection heuristic	p. 39
KALIS_NETWORK_SIMPLEX	Set the solution algorithm for the relaxation to network simplex	p. 55
KALIS_NON_LINEAR_CONSTRAINTS	Set of non linear constraints	p. 49
KALIS_OPTIMAL_SOLUTION	Schedule search strategy choice: improving bound and prove optimality	p. 46
KALIS_PRIMAL_SIMPLEX	Set the solution algorithm for the relaxation to primal simplex	p. 54
KALIS_RANDOM_VALUE	Random value selection heuristic	p. 40
KALIS_RANDOM_VARIABLE	Random variable selection heuristic	p. 37
KALIS_RELAX_ALGORITHM	Set the solution algorithm for the relaxation	p. 53
KALIS_RELAX_GLOBAL	Set global search flag for the linear relaxation	p. 52
KALIS_RELAX_MIP_ABS_STOP	Set the MIP absolute tolerance to optimality for the resolution of the linear relaxation solver	p. 53
KALIS_RELAX_MIP_REL_STOP	Set the MIP relative tolerance to optimality for the resolution of the linear relaxation solver	p. 53
KALIS_RELAX_OPT_TOL	Set the optimality tolerance for the resolution of the linear relaxation solver	p. 53
KALIS_RELAX_PRESOLVE	Use presolve for relaxation	p. 52
KALIS_RELAX_RCOSTS_PROPAG	Use reduced costs for propagation	p. 52
KALIS_RELAX_RELOAD_BASIS	Use reload basis for relaxation	p. 52
KALIS_RESET_OPT_PARAMS	Group parameter choice for strategy optimisation	p. 48
KALIS_RESET_PARAMS_ALL	Group parameter choice for all parameters	p. 46
KALIS_RESET_SEARCH_PARAMS	Group parameter choice for strategy properties	p. 48
KALIS_RESET_VAR_BOUNDS	Group parameter choice for variable bounds	p. 47
KALIS_RESET_VAR_PRECISION	Group parameter choice for variable precisions	p. 47
KALIS_SDOMDEG_RATIO	Smallest domain size to degree ratio variable selection heuristic.	p. 37
KALIS_SLIM_BY_BACKTRACKS	Type of limitation on the search tree	p. 33
KALIS_SLIM_BY_DEPTH	Type of limitation on the search tree	p. 33
KALIS_SLIM_BY_NODES	Type of limitation on the search tree	p. 32

KALIS_SLIM_BY_SOLUTIONS	Type of limitation on the search tree	p. 33
KALIS_SLIM_BY_TIME	Type of limitation on the search tree	p. 33
KALIS_SLIM_UNREACHED	Type of limitation on the search tree	p. 32
KALIS_SMALLEST_DOMAIN	Smallest domain variable selection heuristic (finite domain and continuous variables)	p. 37
KALIS_SMALLEST_ECT	Smallest Earliest Completion time task selection heuristic	p. 41
KALIS_SMALLEST_EST	Smallest Earliest Start time task selection heuristic	p. 41
KALIS_SMALLEST_LCT	Smallest Latest Completion time task selection heuristic	p. 41
KALIS_SMALLEST_LST	Smallest Latest Start time task selection heuristic	p. 42
KALIS_SMALLEST_MAX	Smallest maximum variable selection heuristic	p. 38
KALIS_SMALLEST_MIN	Smallest minimum variable selection heuristic	p. 38
KALIS_SOLVE_AS_LP	Use LP relaxation	p. 51
KALIS_SOLVE_AS_MIP	Use MIP relaxation	p. 51
KALIS_TASK_INPUT_ORDER	Input order task selection heuristic	p. 43
KALIS_TASK_INTERVALS	Task intervals propagation type for resource constraints	p. 45
KALIS_TASK_RANDOM_ORDER	Random order task selection heuristic	p. 43
KALIS_TIMETABLING	Timetabling propagation type for resource constraints	p. 44
KALIS_TLIM_ABS_OPT	Limitation on the optimization through absolute tolerance	p. 34
KALIS_TLIM_REL_OPT	Limitation on the optimization through relative tolerance	p. 34
KALIS_TLIM_UNREACHED	No limitation on the optimization due to tolerance limits	p. 34
KALIS_TOPNODE_RELAX_SOLVER	Top node relaxation solver configuration	p. 48
KALIS_TREENODE_RELAX_SOLVER	Tree node relaxation solver configuration	p. 49
KALIS_UNARY_RESOURCE	Unary resource	p. 40
KALIS_WIDEST_DOMAIN	Widest domain variable selection heuristic (continuous variables)	p. 38

KALIS_FORWARD_CHECKING

Description	Forward checking propagation type for the <code>all_different</code> constraint
Type	Integer, read only
Values	0
Notes	This constant is passed to the <code>all_different</code> constraint to specify the kind of propagation used to filter it. When <code>KALIS_FORWARD_CHECKING</code> is used, the filtering algorithm achieves a very simple, and fast, filtering. Although less powerful than the generalized algorithm (<code>KALIS_GEN_ARC_CONSISTENCY</code>) that may lead to a stronger pruning, the speed of the <code>KALIS_FORWARD_CHECKING</code> filtering allows the search process to explore more nodes in the same amount of time which makes the algorithm competitive for simple problems.
See also	<code>all_different</code> <code>KALIS_GEN_ARC_CONSISTENCY</code>

KALIS_GEN_ARC_CONSISTENCY

Description	Generalized arc consistency propagation type for the all_different constraint
Type	Integer, read only
Values	3
Notes	This constant is passed to the all_different constraint to specify the kind of propagation used to filter it. When KALIS_GEN_ARC_CONSISTENCY is used, the filtering algorithm achieves generalized arc-consistency (cf. [R94]). Although this algorithm may filter more values than the KALIS_FORWARD_CHECKING algorithm, this additional pruning comes at the price of a computational overhead at each node that is not necessary for simple problems.
See also	all_different KALIS_FORWARD_CHECKING

KALIS_SLIM_UNREACHED

Description	Type of limitation on the search tree
Type	Integer, read only
Values	0
Notes	This constant represents the value of the parameter KALIS_SEARCH_LIMIT when no limitation of the search process has occurred.
See also	KALIS_SLIM_BY_NODES KALIS_SLIM_BY_SOLUTIONS KALIS_SLIM_BY_DEPTH KALIS_SLIM_BY_TIME KALIS_SLIM_BY_BACKTRACKS

KALIS_SLIM_BY_NODES

Description	Type of limitation on the search tree
Type	Integer, read only
Values	1
Notes	This constant represents the value of the parameter KALIS_SEARCH_LIMIT when the maximum number of nodes allowed for the search process has been reached.
See also	KALIS_SLIM_BY_NODES KALIS_SLIM_BY_SOLUTIONS KALIS_SLIM_BY_DEPTH KALIS_SLIM_BY_TIME KALIS_SLIM_BY_BACKTRACKS

KALIS_SLIM_BY_SOLUTIONS

Description	Type of limitation on the search tree
Type	Integer, read only
Values	2
Notes	This constant represents the value of the parameter <code>KALIS_SEARCH_LIMIT</code> when the maximum number of solutions to be found during the search process has been reached.
See also	<code>KALIS_SLIM_BY_NODES</code> <code>KALIS_SLIM_BY_SOLUTIONS</code> <code>KALIS_SLIM_BY_DEPTH</code> <code>KALIS_SLIM_BY_TIME</code> <code>KALIS_SLIM_BY_BACKTRACKS</code>

KALIS_SLIM_BY_DEPTH

Description	Type of limitation on the search tree
Type	Integer, read only
Values	3
Notes	This constant represents the value of the parameter <code>KALIS_SEARCH_LIMIT</code> when the maximum depth of the search tree has been reached.
See also	<code>KALIS_SLIM_BY_NODES</code> <code>KALIS_SLIM_BY_SOLUTIONS</code> <code>KALIS_SLIM_BY_DEPTH</code> <code>KALIS_SLIM_BY_TIME</code> <code>KALIS_SLIM_BY_BACKTRACKS</code>

KALIS_SLIM_BY_TIME

Description	Type of limitation on the search tree
Type	Integer, read only
Values	4
Notes	This constant represents the value of the parameter <code>KALIS_SEARCH_LIMIT</code> when the maximum computation time allowed to the search process has been reached.
See also	<code>KALIS_SLIM_BY_NODES</code> <code>KALIS_SLIM_BY_SOLUTIONS</code> <code>KALIS_SLIM_BY_DEPTH</code> <code>KALIS_SLIM_BY_TIME</code> <code>KALIS_SLIM_BY_BACKTRACKS</code>

KALIS_SLIM_BY_BACKTRACKS

Description	Type of limitation on the search tree
Type	Integer, read only

Values	5
Notes	This constant represents the value of the parameter <code>KALIS_SEARCH_LIMIT</code> when the maximum number of backtracks was reached during the search process and the search process was interrupted.
See also	<code>KALIS_SLIM_BY_NODES</code> <code>KALIS_SLIM_BY_SOLUTIONS</code> <code>KALIS_SLIM_BY_DEPTH</code> <code>KALIS_SLIM_BY_TIME</code> <code>KALIS_SLIM_BY_BACKTRACKS</code>

KALIS_TLIM_UNREACHED

Description	No limitation on the optimization due to tolerance limits
Type	Integer, read only
Values	0
Notes	This constant represents the value of the parameter <code>KALIS_TOLERANCE_LIMIT</code> when no limitation of the optimization process has occurred.
See also	<code>KALIS_TLIM_UNREACHED</code> <code>KALIS_TLIM_ABS_OPT</code> <code>KALIS_TLIM_REL_OPT</code>

KALIS_TLIM_ABS_OPT

Description	Limitation on the optimization through absolute tolerance
Type	Integer, read only
Values	1
Notes	This constant represents the value of the parameter <code>KALIS_TOLERANCE_LIMIT</code> when the search process has been interrupted due to an absolute tolerance limit.
See also	<code>KALIS_TLIM_UNREACHED</code> <code>KALIS_TLIM_ABS_OPT</code> <code>KALIS_TLIM_REL_OPT</code>

KALIS_TLIM_REL_OPT

Description	Limitation on the optimization through relative tolerance
Type	Integer, read only
Values	2
Notes	This constant represents the value of the parameter <code>KALIS_TOLERANCE_LIMIT</code> when the search process has been interrupted due to a relative tolerance limit
See also	<code>KALIS_TLIM_UNREACHED</code> <code>KALIS_TLIM_ABS_OPT</code>

KALIS_INPUT_ORDER

Description	Input Order variable selection heuristic
Type	String, read only
Values	*a
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The input order heuristic selects non-instantiated variables with the smallest index for indexed containers and in the order of iteration for ordered containers such as lists and arrays.
See also	<code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_MAX_DEGREE

Description	Maximum constraint graph degree variable selection heuristic
Type	String, read only
Values	*b
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The max degree heuristic selects non-instantiated variables with the largest degree in the constraint graph (<i>i.e.</i> , that are involved in the maximum number of different constraints).
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_LARGEST_MIN

Description	Largest minimum variable selection heuristic
Type	String, read only
Values	*c
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The largest min heuristic selects non-instantiated variables with the largest lower bound.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_LARGEST_MAX

Description	Largest maximum variable selection heuristic
Type	String, read only
Values	*d
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The largest maximum heuristic selects non-instantiated variables with the largest upper bound.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_MAXREGRET_LB

Description	Maximum regret on the lower bound variable selection heuristic
Type	String, read only
Values	*e
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. This variable selection will select the variable maximizing the distance between its lower bound and its second-smallest value.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_MAXREGRET_UB

Description	Maximum regret on the upper bound variable selection heuristic
Type	String, read only
Values	*f
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. This variable selection will select the variable maximizing the distance between its upper bound and the second-largest value.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_RANDOM_VARIABLE

Description	Random variable selection heuristic
Type	String, read only
Values	*g
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The random variable heuristic selects non-instantiated variables in a random order.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_SDOMDEG_RATIO

Description	Smallest domain size to degree ratio variable selection heuristic.
Type	String, read only
Values	*h
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. This heuristic selects the variable having the smallest domain size to degree ratio. The degree of a variable is defined as the number of constraints using this variable.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_SMALLEST_DOMAIN

Description	Smallest domain variable selection heuristic (finite domain and continuous variables)
Type	String, read only
Values	*i
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The smallest domain heuristic selects non instantiated variables with the smallest domain size (number of distinct values in the domain).
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code> <code>KALIS_WIDEST_DOMAIN</code>

KALIS_SMALLEST_MAX

Description	Smallest maximum variable selection heuristic
Type	String, read only
Values	*j
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The smallest maximum heuristic selects non instantiated variables with the smallest upper bound.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_SMALLEST_MIN

Description	Smallest minimum variable selection heuristic
Type	String, read only
Values	*k
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the variable selection heuristic. The smallest minimum heuristic selects non instantiated variables with the smallest lower bound.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code>

KALIS_WIDEST_DOMAIN

Description	Widest domain variable selection heuristic (continuous variables)
Type	String, read only
Values	*l
Notes	Passes this constant to the constructor of branching schemes for continuous decision variables (<code>split_domain</code>) to specify the variable selection heuristic. The widest domain heuristic selects non instantiated variables with the largest value interval.
See also	<code>KALIS_INPUT_ORDER</code> <code>KALIS_MAX_DEGREE</code> <code>KALIS_LARGEST_MIN</code> <code>KALIS_LARGEST_MAX</code> <code>KALIS_MAXREGRET_LB</code> <code>KALIS_MAXREGRET_UB</code> <code>KALIS_RANDOM_VARIABLE</code> <code>KALIS_SDOMDEG_RATIO</code> <code>KALIS_SMALLEST_DOMAIN</code> <code>KALIS_SMALLEST_MAX</code> <code>KALIS_SMALLEST_MIN</code>

KALIS_MAX_TO_MIN

Description	Maximum to minimum value selection heuristic
Type	String, read only
Values	+1
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the value selection heuristic. The max to min heuristic selects values in decreasing order.
See also	KALIS_MIN_TO_MAX KALIS_MIDDLE_VALUE KALIS_NEAREST_VALUE KALIS_RANDOM_VALUE

KALIS_MIN_TO_MAX

Description	Minimum to maximum value selection heuristic
Type	String, read only
Values	+2
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the value selection heuristic. The min to max heuristic selects values in increasing order.
See also	KALIS_MAX_TO_MIN KALIS_MIDDLE_VALUE KALIS_NEAREST_VALUE KALIS_RANDOM_VALUE

KALIS_MIDDLE_VALUE

Description	Middle value selection heuristic
Type	String, read only
Values	+3
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the value selection heuristic. The middle value heuristic selects the value in the domain of a variable that minimizes the distance to the median value of the domain.
See also	KALIS_MAX_TO_MIN KALIS_MIN_TO_MAX KALIS_NEAREST_VALUE KALIS_RANDOM_VALUE

KALIS_NEAREST_VALUE

Description	Nearest to target value selection heuristic
Type	String, read only

Values	+4
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the value selection heuristic. The nearest value heuristic selects the value in the domain of a variable that minimizes the distance to the target value of the variable.
See also	<code>KALIS_MAX_TO_MIN</code> <code>KALIS_MIN_TO_MAX</code> <code>KALIS_MIDDLE_VALUE</code> <code>KALIS_RANDOM_VALUE</code>

KALIS_RANDOM_VALUE

Description	Random value selection heuristic
Type	String, read only
Values	+5
Notes	Passes this constant to the constructor of branching schemes (for example <code>assign_var</code>) to specify the value selection heuristic. The random value heuristic selects a random value in the domain of a variable.
See also	<code>KALIS_MAX_TO_MIN</code> <code>KALIS_MIN_TO_MAX</code> <code>KALIS_MIDDLE_VALUE</code> <code>KALIS_NEAREST_VALUE</code>

KALIS_COPYRIGHT

Description	Xpress Kalis copyright information
Type	String, read only
Values	(c) Copyrights Artelys S.A. 2000-2013

KALIS_UNARY_RESOURCE

Description	Unary resource
Type	Integer, read only
Values	0
Notes	Passes this constant to the <code>set_resource_attributes</code> subroutine to specify a unary resource (a resource that can be used only by one task at a time).
See also	<code>KALIS_DISCRETE_RESOURCE</code>

KALIS_DISCRETE_RESOURCE

Description	Discrete resource
Type	Integer, read only

Values	1
Notes	Passes this constant to the <code>set_resource_attributes</code> subroutine to specify a discrete resource (a resource than can be used by several tasks at the same time).
See also	<code>KALIS_UNARY_RESOURCE</code>

KALIS_SMALLEST_ECT

Description	Smallest Earliest Completion time task selection heuristic
Type	String, read only
Values	\$1
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_SMALLEST_EST

Description	Smallest Earliest Start time task selection heuristic
Type	String, read only
Values	\$2
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_SMALLEST_LCT

Description	Smallest Latest Completion time task selection heuristic
Type	String, read only
Values	\$3
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_SMALLEST_LST

Description	Smallest Latest Start time task selection heuristic
Type	String, read only
Values	\$4
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	KALIS_SMALLEST_ECT KALIS_SMALLEST_EST KALIS_SMALLEST_LCT KALIS_SMALLEST_LST KALIS_LARGEST_ECT KALIS_LARGEST_EST KALIS_LARGEST_LCT KALIS_LARGEST_LST

KALIS_LARGEST_ECT

Description	Largest Earliest Completion time task selection heuristic
Type	String, read only
Values	\$5
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	KALIS_SMALLEST_ECT KALIS_SMALLEST_EST KALIS_SMALLEST_LCT KALIS_SMALLEST_LST KALIS_LARGEST_ECT KALIS_LARGEST_EST KALIS_LARGEST_LCT KALIS_LARGEST_LST

KALIS_LARGEST_EST

Description	Largest Earliest Start time task selection heuristic
Type	String, read only
Values	\$6
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	KALIS_SMALLEST_ECT KALIS_SMALLEST_EST KALIS_SMALLEST_LCT KALIS_SMALLEST_LST KALIS_LARGEST_ECT KALIS_LARGEST_EST KALIS_LARGEST_LCT KALIS_LARGEST_LST

KALIS_LARGEST_LCT

Description	Largest Latest Completion time task selection heuristic
Type	String, read only

Values	\$7
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_LARGEST_LST

Description	Largest Latest Start time task selection heuristic
Type	String, read only
Values	\$8
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_TASK_INPUT_ORDER

Description	Input order task selection heuristic
Type	String, read only
Values	\$9
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_TASK_RANDOM_ORDER</code> <code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_TASK_RANDOM_ORDER

Description	Random order task selection heuristic
Type	String, read only
Values	\$a
Notes	Passes this constant to the constructor of <code>task_serialize</code> to specify the task selection heuristic.
See also	<code>KALIS_TASK_INPUT_ORDER</code> <code>KALIS_SMALLEST_ECT</code> <code>KALIS_SMALLEST_EST</code> <code>KALIS_SMALLEST_LCT</code> <code>KALIS_SMALLEST_LST</code> <code>KALIS_LARGEST_ECT</code> <code>KALIS_LARGEST_EST</code> <code>KALIS_LARGEST_LCT</code> <code>KALIS_LARGEST_LST</code>

KALIS_DISJ_INPUT_ORDER

Description	Input order disjunction selection heuristic
Type	String, read only
Values	%0
Notes	Passes this constant to the constructor of <code>settle_disjunction</code> or <code>probe_settle_disjunction</code> to specify the disjunction selection heuristic.
See also	<code>KALIS_DISJ_PRIORITY_ORDER</code>

KALIS_DISJ_PRIORITY_ORDER

Description	Input order disjunction selection heuristic
Type	String, read only
Values	%1
Notes	Passes this constant to the constructor of <code>settle_disjunction</code> or <code>probe_settle_disjunction</code> to specify the disjunction selection heuristic.
See also	<code>KALIS_DISJ_INPUT_ORDER</code>

KALIS_TIMETABLING

Description	Timetabling propagation type for resource constraints
Type	Integer, read only
Values	1
Notes	This constant is passed to the <code>set_resource_attributes</code> function to specify the kind of propagation to be used for filtering the resource constraint. The KALIS_TIMETABLING filtering algorithm achieves a very simple and fast filtering. Although less powerful than the algorithms KALIS_DISJUNCTIONS (unary resources only) and KALIS_TASK_INTERVALS that may lead to a stronger pruning, the speed of the KALIS_TIMETABLING filtering allows the search process to explore more nodes in the same amount of time which makes the algorithm competitive for simple problems and also for very large problems where the computational overhead of other filtering algorithms is prohibitive.
See also	<code>KALIS_TASK_INTERVALS</code> <code>KALIS_DISJUNCTIONS</code> <code>KALIS_EDGE_FINDING</code>

KALIS_TASK_INTERVALS

Description	Task intervals propagation type for resource constraints
Type	Integer, read only
Values	2
Notes	This constant is passed to the <code>set_resource_attributes</code> procedure to specify the type of propagation to be used for filtering the resource constraint. The KALIS_TASK_INTERVALS filtering algorithm achieves a very strong and relatively slow filtering. Although more powerful than the simple algorithm (KALIS_TIMETABLING), the computational cost of the KALIS_TASK_INTERVALS filtering makes it competitive for medium sized hard problems but is prohibitive for large problems with many tasks per resource (cf. [CL94a], [CL94b]).
See also	KALIS_TIMETABLING KALIS_DISJUNCTIONS KALIS_EDGE_FINDING

KALIS_DISJUNCTIONS

Description	Disjunctions propagation type for unary resource constraints
Type	Integer, read only
Values	4
Notes	This constant is passed to the <code>set_resource_attributes</code> procedure to specify the type of propagation to be used for filtering the resource constraint for unary resources. With KALIS_DISJUNCTIONS the resource constraint will be implemented as $n*(n-1)/2$ disjunctions between pairs of tasks. This algorithm is more powerful than the KALIS_TIMETABLING algorithm but less powerful than the algorithm KALIS_TASK_INTERVALS that may lead to a stronger pruning at the cost of additional computation overhead. This algorithm is generally competitive for small to medium size problems where the number of tasks using the resource is not overly large.
See also	KALIS_TASK_INTERVALS KALIS_TIMETABLING KALIS_EDGE_FINDING

KALIS_EDGE_FINDING

Description	Tasks Interval + Edge finding propagation type for discrete resource constraints
Type	Integer, read only
Values	4
Notes	This constant is passed to the <code>set_resource_attributes</code> procedure to specify the kind of propagation used to filter the resource constraint. When KALIS_EDGE_FINDING is used, the resource constraint will add Edge finding propagation rules to the Tasks Interval propagation scheme. This algorithm is more powerful than KALIS_TASK_INTERVALS algorithm and may lead to a stronger pruning at the cost of additional computation

overhead. This algorithm is generally competitive for small to medium size problems where the number of tasks using the resource is not too large (cf. [LP96], [MVH08]).

See also [KALIS_TASK_INTERVALS](#) [KALIS_TIMETABLING](#) [KALIS_DISJUNCTIONS](#)

KALIS_INITIAL_SOLUTION

Description	Schedule search strategy choice: heuristic solution
Type	Integer, read only
Values	0
Notes	This constant is passed to the cp_set_schedule_strategy subroutine to specify the phase of the algorithm. KALIS_INITIAL_SOLUTION denotes the first phase of the algorithm and KALIS_OPTIMAL_SOLUTION selects the second phase of the algorithm.
See also	KALIS_OPTIMAL_SOLUTION cp_schedule

KALIS_OPTIMAL_SOLUTION

Description	Schedule search strategy choice: improving bound and prove optimality
Type	Integer, read only
Values	1
Notes	This constant is passed to the cp_set_schedule_strategy subroutine to specify the phase of the algorithm. KALIS_INITIAL_SOLUTION denotes the first phase of the algorithm and KALIS_OPTIMAL_SOLUTION selects the second phase of the algorithm.
See also	KALIS_INITIAL_SOLUTION cp_schedule

KALIS_RESET_PARAMS_ALL

Description	Group parameter choice for all parameters
Type	Integer, read only
Values	0
Notes	This constant is passed to the cp_reset_params procedure to specify the group of control parameters. KALIS_RESET_PARAMS_ALL resets all variables and strategy parameters: ■ KALIS_DEFAULT_LB ■ KALIS_DEFAULT_UB ■ KALIS_DEFAULT_CONTINUOUS_LB ■ KALIS_DEFAULT_CONTINUOUS_UB ■ KALIS_DEFAULT_PRECISION_VALUE

- KALIS_DEFAULT_PRECISION_RELATIVITY
 - KALIS_AUTO_PROPAGATE
 - KALIS_OPTIMIZE_WITH_RESTART
 - KALIS_DICHOTOMIC_OBJ_SEARCH
 - KALIS_USE_3B_CONSISTENCY
 - KALIS_VERBOSE_LEVEL
 - KALIS_MAX_NODES
 - KALIS_MAX_SOLUTIONS
 - KALIS_MAX_DEPTH
 - KALIS_OPT_ABS_TOLERANCE
 - KALIS_OPT_REL_TOLERANCE
 - KALIS_MAX_BACKTRACKS
 - KALIS_MAX_COMPUTATION_TIME
 - KALIS_MAX_NODES_BETWEEN_SOLUTIONS
-

KALIS_RESET_VAR_BOUNDS

Description	Group parameter choice for variable bounds
Type	Integer, read only
Values	1
Notes	This constant is passed to the <code>cp_reset_params</code> procedure to specify the group of control parameters. KALIS_RESET_VAR_BOUNDS resets bounds of discrete and continuous variables: ■ KALIS_DEFAULT_LB ■ KALIS_DEFAULT_UB ■ KALIS_DEFAULT_CONTINUOUS_LB ■ KALIS_DEFAULT_CONTINUOUS_UB

KALIS_RESET_VAR_PRECISION

Description	Group parameter choice for variable precisions
Type	Integer, read only
Values	2
Notes	This constant is passed to the <code>cp_reset_params</code> function to specify the group of control parameters. KALIS_RESET_VAR_PRECISION resets the precision for continuous variables: ■ KALIS_DEFAULT_PRECISION_VALUE ■ KALIS_DEFAULT_PRECISION_RELATIVITY

KALIS_RESET_OPT_PARAMS

Description	Group parameter choice for strategy optimisation
Type	Integer, read only
Values	3
Notes	This constant is passed to the <code>cp_reset_params</code> procedure to specify the group of control parameters. KALIS_RESET_OPT_PARAMS resets mechanisms of search: ■ KALIS_AUTO_PROPAGATE ■ KALIS_OPTIMIZE_WITH_RESTART ■ KALIS_DICHOTOMIC_OBJ_SEARCH ■ KALIS_USE_3B_CONSISTENCY

KALIS_RESET_SEARCH_PARAMS

Description	Group parameter choice for strategy properties
Type	Integer, read only
Values	4
Notes	This constant is passed to the <code>cp_reset_params</code> procedure to specify the group of control parameters. KALIS_RESET_SEARCH_PARAMS resets properties of search : ■ KALIS_VERBOSE_LEVEL ■ KALIS_MAX_NODES ■ KALIS_MAX_SOLUTIONS ■ KALIS_MAX_DEPTH ■ KALIS_OPT_ABS_TOLERANCE ■ KALIS_OPT_REL_TOLERANCE ■ KALIS_MAX_BACKTRACKS ■ KALIS_MAX_COMPUTATION_TIME ■ KALIS_MAX_NODES_BETWEEN_SOLUTIONS

KALIS_TOPNODE_RELAX_SOLVER

Description	Top node relaxation solver configuration
Type	Integer, read only
Values	200
Notes	This constant is passed to the <code>get_linrelax_solver</code> function to specify the predefined relaxation solver configuration. In this configuration, the relaxation is solved only once at the top node of the search tree.
See also	<code>KALIS_BILEVEL_RELAX_SOLVER</code> <code>KALIS_TREENODE_RELAX_SOLVER</code>

KALIS_TREENODE_RELAX_SOLVER

Description	Tree node relaxation solver configuration
Type	Integer, read only
Values	201
Notes	This constant is passed to the <code>get_linrelax_solver</code> function to specify the predefined relaxation solver configuration. In this configuration, the relaxation is solved at all nodes (except the top node) of the search tree.
See also	<code>KALIS_TOPNODE_RELAX_SOLVER</code> <code>KALIS_BILEVEL_RELAX_SOLVER</code>

KALIS_BILEVEL_RELAX_SOLVER

Description	Bilevel node relaxation solver configuration
Type	Integer, read only
Values	202
Notes	This constant is passed to the <code>get_linrelax_solver</code> function to specify the predefined relaxation solver configuration. In this configuration, the relaxation is solved only when a user defined set of variables are instantiated (by default all the cpfloatvars).
See also	<code>KALIS_TOPNODE_RELAX_SOLVER</code> <code>KALIS_TREENODE_RELAX_SOLVER</code>

KALIS_LINEAR_CONSTRAINTS

Description	Set of linear constraints
Type	Integer, read only
Values	0
Notes	This constant is passed to the <code>cp_get_linrelax</code> function to specify the type of constraints to be relaxed.
See also	<code>KALIS_NON_LINEAR_CONSTRAINTS</code> <code>KALIS_LOGICAL_CONSTRAINTS</code> <code>KALIS_DISTANCE_CONSTRAINTS</code> <code>KALIS_ALL_CONSTRAINTS</code> <code>cp_get_linrelax</code>

KALIS_NON_LINEAR_CONSTRAINTS

Description	Set of non linear constraints
Type	Integer, read only

Values	1
Notes	This constant is passed to the <code>cp_get_linrelax</code> function to specify the type of constraints to be relaxed.
See also	<code>KALIS_LINEAR_CONSTRAINTS</code> <code>KALIS_LOGICAL_CONSTRAINTS</code> <code>KALIS_DISTANCE_CONSTRAINTS</code> <code>KALIS_ALL_CONSTRAINTS</code> <code>cp_get_linrelax</code>

KALIS_LOGICAL_CONSTRAINTS

Description	Set of logical constraints
Type	Integer, read only
Values	2
Notes	This constant is passed to the <code>cp_get_linrelax</code> function to specify the type of constraints to be relaxed.
See also	<code>KALIS_LINEAR_CONSTRAINTS</code> <code>KALIS_NON_LINEAR_CONSTRAINTS</code> <code>KALIS_DISTANCE_CONSTRAINTS</code> <code>KALIS_ALL_CONSTRAINTS</code> <code>cp_get_linrelax</code>

KALIS_DISTANCE_CONSTRAINTS

Description	Set of the distance constraints
Type	Integer, read only
Values	3
Notes	This constant is passed to the <code>cp_get_linrelax</code> function to specify the type of constraints to be relaxed.
See also	<code>KALIS_LINEAR_CONSTRAINTS</code> <code>KALIS_NON_LINEAR_CONSTRAINTS</code> <code>KALIS_LOGICAL_CONSTRAINTS</code> <code>KALIS_ALL_CONSTRAINTS</code> <code>cp_get_linrelax</code>

KALIS_ALL_CONSTRAINTS

Description	Set of all the constraints
Type	Integer, read only
Values	4
Notes	This constant is passed to the <code>cp_get_linrelax</code> function to specify the type of constraints to be relaxed.
See also	<code>KALIS_LINEAR_CONSTRAINTS</code> <code>KALIS_NON_LINEAR_CONSTRAINTS</code> <code>KALIS_LOGICAL_CONSTRAINTS</code> <code>KALIS_DISTANCE_CONSTRAINTS</code> <code>cp_get_linrelax</code>

KALIS_MINIMIZE

Description	Minimize objective in linear relaxation
Type	Integer, read only
Values	100
Notes	This constant is passed to the <code>cp_set_linrelax</code> function to specify the sense of objective.

KALIS_MAXIMIZE

Description	Maximize objective in linear relaxation
Type	Integer, read only
Values	101
Notes	This constant is passed to the <code>cp_set_linrelax</code> function to specify the sense of objective.

KALIS_SOLVE_AS_LP

Description	Use LP relaxation
Type	Integer, read only
Values	10
Notes	This constant is passed to the <code>get_linrelax_solver</code> function to specify the type of method use.

KALIS_SOLVE_AS_MIP

Description	Use MIP relaxation
Type	Integer, read only
Values	11
Notes	This constant is passed to the <code>get_linrelax_solver</code> function to specify the type of method use.

KALIS_RELAX_PRESOLVE

Description	Use presolve for relaxation
Type	Integer, read only
Values	10001
Notes	This constant is passed to the <code>set_linrelax_solver_attribute</code> procedure to select the parameter to set. Possible settings: 1 — use presolve, 0 — presolve off.

KALIS_RELAX_RCOSTS_PROPAG

Description	Use reduced costs for propagation
Type	Integer, read only
Values	10002
Notes	This constant is passed to the <code>set_linrelax_solver_attribute</code> procedure to select the parameter to set. Possible settings: 1 — use reduced cost information, 0 — do not use reduced costs.

KALIS_RELAX_RELOAD_BASIS

Description	Use reload basis for relaxation
Type	Integer, read only
Values	10003
Notes	This constant is passed to the <code>set_linrelax_solver_attribute</code> procedure to select the parameter to set. Possible settings: 1 — save/reload basis, 0 — do not load any basis.

KALIS_RELAX_GLOBAL

Description	Set global search flag for the linear relaxation
Type	Integer, read only
Values	10004
Notes	This constant is passed to the <code>set_linrelax_solver_attribute</code> procedure to select the parameter to set. Possible settings: 1 — treat problem as MIP, 0 — treat problem as LP (<i>i.e.</i> , ignore any MIP information).

KALIS_RELAX_ALGORITHM

Description	Set the solution algorithm for the relaxation
Type	Integer, read only
Values	10005
Notes	This constant is passed to the <code>set_linrelax_solver_attribute</code> procedure to select the solution algorithm, possible values for this parameter: <code>KALIS_PRIMAL_SIMPLEX</code> , <code>KALIS_DUAL_SIMPLEX</code> , <code>KALIS_BARRIER</code> , <code>KALIS_NETWORK_SIMPLEX</code>
See also	<code>KALIS_PRIMAL_SIMPLEX</code> <code>KALIS_DUAL_SIMPLEX</code> <code>KALIS_BARRIER</code> <code>KALIS_NETWORK_SIMPLEX</code>

KALIS_RELAX_OPT_TOL

Description	Set the optimality tolerance for the resolution of the linear relaxation solver
Type	Integer, read only
Values	10010
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_OPT_TOL</code> parameter.
See also	<code>KALIS_RELAX_OPT_TOL</code>

KALIS_RELAX_MIP_REL_STOP

Description	Set the MIP relative tolerance to optimality for the resolution of the linear relaxation solver
Type	Integer, read only
Values	10011
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_MIP_REL_STOP</code> parameter.
See also	<code>KALIS_RELAX_MIP_ABS_STOP</code>

KALIS_RELAX_MIP_ABS_STOP

Description	Set the MIP absolute tolerance to optimality for the resolution of the linear relaxation solver
Type	Integer, read only

Values	10012
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_MIP_ABS_STOP</code> parameter.
See also	<code>KALIS_RELAX_MIP_REL_STOP</code>

KALIS_PRIMAL_SIMPLEX

Description	Set the solution algorithm for the relaxation to primal simplex
Type	Integer, read only
Values	20001
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_ALGORITHM</code> parameter.
See also	<code>KALIS_RELAX_ALGORITHM</code>

KALIS_DUAL_SIMPLEX

Description	Set the solution algorithm for the relaxation to dual simplex
Type	Integer, read only
Values	20002
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_ALGORITHM</code> parameter.
See also	<code>KALIS_RELAX_ALGORITHM</code>

KALIS_BARRIER

Description	Set the solution algorithm for the relaxation to barrier
Type	Integer, read only
Values	20003
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_ALGORITHM</code> parameter.
See also	<code>KALIS_RELAX_ALGORITHM</code>

KALIS_NETWORK_SIMPLEX

Description	Set the solution algorithm for the relaxation to network simplex
Type	Integer, read only
Values	20004
Notes	This constant is passed to the procedure <code>set_linrelax_solver_attribute</code> as value for the <code>KALIS_RELAX_ALGORITHM</code> parameter.
See also	<code>KALIS_RELAX_ALGORITHM</code>

CHAPTER 8

Parameters

KALIS_AUTO_PROPAGATE	Propagates automatically	p. 57
KALIS_AUTO_RELAX	Use linear relaxation automatically	p. 64
KALIS_BACKTRACKS	Number of backtracks	p. 59
KALIS_CHECK_SOLUTION	Automated solution checking	p. 62
KALIS_COMPUTATION_TIME	Computation time	p. 59
KALIS_DEFAULT_CONTINUOUS_LB	Default lower bound for domains of cpfloatvar	p. 62
KALIS_DEFAULT_CONTINUOUS_UB	Default upper bound for domains of cpfloatvar	p. 62
KALIS_DEFAULT_LB	Default lower bound for finite domain variables	p. 57
KALIS_DEFAULT_PRECISION_RELATIVITY	Precision relativity for continuous variables	p. 63
KALIS_DEFAULT_PRECISION_VALUE	Default precision for continuous variables	p. 63
KALIS_DEFAULT_SCHEDULE_HORIZ_MAX	Default maximal time horizon for schedules	p. 63
KALIS_DEFAULT_SCHEDULE_HORIZ_MIN	Default minimal time horizon for schedules	p. 63
KALIS_DEFAULT_UB	Default upper bound for finite domain variables	p. 57
KALIS_DEPTH	Depth of tree search	p. 58
KALIS_DICHOTOMIC_OBJ_SEARCH	Uses a dichotomic strategy to find the optimal objective value	p. 64
KALIS_MAX_BACKTRACKS	Maximum number of backtracks	p. 60
KALIS_MAX_COMPUTATION_TIME	Maximum computation time	p. 61
KALIS_MAX_DEPTH	Maximum depth	p. 60
KALIS_MAX_NODES	Maximum number of nodes	p. 59
KALIS_MAX_NODES_BETWEEN_SOLUTIONS	Number of nodes between two solutions	p. 61
KALIS_MAX_SOLUTIONS	Maximum number of solutions	p. 60
KALIS_NARY_OBJ_SEARCH	Uses an n-ary search strategy to find the optimal objective value	p. 65
KALIS_NODES	Counter for number of nodes	p. 58
KALIS_OPT_ABS_TOLERANCE	Absolute optimality tolerance	p. 61
KALIS_OPT_REL_TOLERANCE	Relative optimality tolerance	p. 61

<code>KALIS_OPTIMIZE_WITH_RESTART</code>	Whether to restart search after each solution	p. 58
<code>KALIS_SCHEDULE_ENABLE_SHAVING</code>	Enable shaving during initial step of schedule search	p. 65
<code>KALIS_SEARCH_LIMIT</code>	Search limit reached	p. 58
<code>KALIS_THREADS</code>	Number of threads	p. 65
<code>KALIS_TOLERANCE_LIMIT</code>	Tolerance limit reached	p. 59
<code>KALIS_USE_3B_CONSISTENCY</code>	Activates 3B consistency for continuous variables	p. 64
<code>KALIS_VERBOSE_LEVEL</code>	Verbosity level of Xpress Kalis	p. 64

KALIS_DEFAULT_LB

Description	Default lower bound for finite domain variables
Type	Integer, read/write
Values	-10000 default value
Notes	Sets the default lower bound for the initial domain of enumerated domain decision variables.
See also	<code>KALIS_DEFAULT_UB</code> <code>getlb</code>

KALIS_DEFAULT_UB

Description	Default upper bound for finite domain variables
Type	Integer, read/write
Values	+10000 default value
Notes	Sets the default upper bound for the initial domain of enumerated domain decision variables.
See also	<code>KALIS_DEFAULT_LB</code> <code>getub</code>

KALIS_AUTO_PROPAGATE

Description	Propagates automatically
Type	Boolean, read/write
Values	true default value
Notes	Enables propagating constraints instead of posting them or searching for solutions. Note that domain definition of decision variables is always taken into account irrespective of whether this control is on or off.
See also	<code>cp_propagate</code> <code>cp_post</code>

KALIS_OPTIMIZE_WITH_RESTART

Description	Decides whether to restart the search after each solution or to continue search from the present point with the new bound on the objective.
Type	Boolean, read/write
Values	false default value (no restart)
Affects routines	cp_minimize cp_maximize

KALIS_NODES

Description	Counter for number of nodes
Type	Integer, read only
Notes	Current number of nodes explored by the branch and bound search. This parameter does not work with <code>cp_schedule</code> .
See also	KALIS_MAX_NODES KALIS_DEPTH KALIS_BACKTRACKS KALIS_COMPUTATION_TIME

KALIS_DEPTH

Description	Depth of tree search
Type	Integer, read only
Notes	Maximal depth of the search tree reached by the branch and bound search. This parameter does not work with <code>cp_schedule</code> .
See also	KALIS_MAX_DEPTH KALIS_NODES KALIS_BACKTRACKS KALIS_COMPUTATION_TIME

KALIS_SEARCH_LIMIT

Description	Search limit reached
Type	Integer, read only
Notes	Active search limit on the search process if any
See also	KALIS_SLIM_UNREACHED KALIS_SLIM_BY_NODES KALIS_SLIM_BY_SOLUTIONS KALIS_SLIM_BY_DEPTH KALIS_SLIM_BY_TIME KALIS_SLIM_BY_BACKTRACKS

KALIS_TOLERANCE_LIMIT

Description	Tolerance limit reached
Type	Integer, read only
Notes	Active tolerance limit on the search process if any
See also	KALIS_TLIM_UNREACHED KALIS_TLIM_ABS_OPT KALIS_TLIM_REL_OPT KALIS_OPT_REL_TOLERANCE KALIS_OPT_ABS_TOLERANCE

KALIS_BACKTRACKS

Description	Number of backtracks
Type	Integer, read only
Notes	Current number of backtracks that occurred during the branch and bound search.
See also	KALIS_MAX_SOLUTIONS KALIS_MAX_NODES KALIS_MAX_COMPUTATION_TIME KALIS_MAX_DEPTH KALIS_MAX_BACKTRACKS KALIS_MAX_NODES_BETWEEN_SOLUTIONS

KALIS_COMPUTATION_TIME

Description	Computation time
Type	Double, read only
Notes	Current computation time (in seconds) of the search process.
See also	KALIS_MAX_SOLUTIONS KALIS_MAX_NODES KALIS_MAX_COMPUTATION_TIME KALIS_MAX_DEPTH KALIS_MAX_BACKTRACKS KALIS_MAX_NODES_BETWEEN_SOLUTIONS

KALIS_MAX_NODES

Description	Maximum number of nodes
Type	Integer, read/write
Values	-1 default value (no limit)
Notes	Limits the number of nodes explored during the branch and bound tree search.
Affects routines	cp_find_next_sol cp_minimize cp_maximize
See also	KALIS_NODES KALIS_MAX_SOLUTIONS KALIS_MAX_COMPUTATION_TIME KALIS_MAX_DEPTH KALIS_MAX_BACKTRACKS KALIS_MAX_NODES_BETWEEN_SOLUTIONS

KALIS_MAX_SOLUTIONS

Description	Maximum number of solutions
Type	Integer, read/write
Values	-1 default value (no limit)
Notes	Limits the number of solutions to be found during the branch and bound tree search.
Affects routines	<code>cp_find_next_sol</code> <code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_MAX_NODES</code> <code>KALIS_MAX_COMPUTATION_TIME</code> <code>KALIS_MAX_DEPTH</code> <code>KALIS_MAX_BACKTRACKS</code> <code>KALIS_MAX_NODES_BETWEEN_SOLUTIONS</code>

KALIS_MAX_DEPTH

Description	Maximum depth
Type	Integer, read/write
Values	-1 default value (no limit) <code>n</code> >0 maximum depth limited to <code>n</code> levels.
Notes	Limits the depth of the search tree during the branch and bound search. This parameter does not work with <code>cp_schedule</code> .
Affects routines	<code>cp_find_next_sol</code> <code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_DEPTH</code> <code>KALIS_MAX_SOLUTIONS</code> <code>KALIS_MAX_NODES</code> <code>KALIS_MAX_COMPUTATION_TIME</code> <code>KALIS_MAX_BACKTRACKS</code> <code>KALIS_MAX_NODES_BETWEEN_SOLUTIONS</code>

KALIS_MAX_BACKTRACKS

Description	Maximum number of backtracks
Type	Integer, read/write
Values	-1 default value (no limit)
Notes	Limits the number of backtracks that can occur during the search process before termination
Affects routines	<code>cp_find_next_sol</code> <code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_BACKTRACKS</code> <code>KALIS_MAX_SOLUTIONS</code> <code>KALIS_MAX_NODES</code> <code>KALIS_MAX_COMPUTATION_TIME</code> <code>KALIS_MAX_DEPTH</code> <code>KALIS_MAX_NODES_BETWEEN_SOLUTIONS</code>

KALIS_MAX_COMPUTATION_TIME

Description	Maximum computation time
Type	Double, read/write
Values	-1 default value (no limit)
Notes	Limits the maximum computation time (in seconds) allowed to the branch and bound
Affects routines	<code>cp_find_next_sol</code> <code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_COMPUTATION_TIME</code> <code>KALIS_MAX_SOLUTIONS</code> <code>KALIS_MAX_NODES</code> <code>KALIS_MAX_DEPTH</code> <code>KALIS_MAX_BACKTRACKS</code>

KALIS_MAX_NODES_BETWEEN_SOLUTIONS

Description	Number of nodes between two solutions
Type	Integer, read/write
Notes	Maximum number of nodes explored between two solutions. With <code>cp_schedule</code> the limit only applies to the last phase of the solution algorithm.
Affects routines	<code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_NODES</code> <code>KALIS_MAX_NODES</code> <code>KALIS_MAX_SOLUTIONS</code> <code>KALIS_MAX_COMPUTATION_TIME</code> <code>KALIS_MAX_DEPTH</code> <code>KALIS_MAX_BACKTRACKS</code>

KALIS_OPT_ABS_TOLERANCE

Description	Absolute optimality tolerance
Type	Integer, read/write
Values	0 default value (no tolerance)
Notes	Enables the search process to stop when the distance between the current solution and the overall optimum is ensured to be less than <code>KALIS_OPT_ABS_TOLERANCE</code> .
Affects routines	<code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_OPT_REL_TOLERANCE</code>

KALIS_OPT_REL_TOLERANCE

Description	Relative optimality tolerance
Type	Double, read/write

Values	0.0 default value (no tolerance)
Notes	Enables the search process to stop when the relative distance between the current solution and the overall optimum solution is ensured to be less than KALIS_OPT_REL_TOLERANCE.
Affects routines	<code>cp_minimize</code> <code>cp_maximize</code>
See also	<code>KALIS_OPT_ABS_TOLERANCE</code>

KALIS_CHECK_SOLUTION

Description	Automated solution checking
Type	Integer, read/write
Values	<code>false</code> default value (no solution checking)
Notes	Enables the automated checking that the solution found is really a solution
Affects routines	<code>cp_find_next_sol</code> <code>cp_minimize</code> <code>cp_maximize</code>

KALIS_DEFAULT_CONTINUOUS_LB

Description	Default lower bound for domains of <code>cpfloatvar</code>
Type	Double, read/write
Values	<code>-1000000</code> default value
Notes	Sets the default lower bound for the initial domain of continuous decision variables
See also	<code>KALIS_DEFAULT_CONTINUOUS_UB</code> <code>getlb</code>

KALIS_DEFAULT_CONTINUOUS_UB

Description	Default upper bound for domains of <code>cpfloatvar</code>
Type	Double, read/write
Values	<code>+1000000</code> default value
Notes	Sets the default upper bound for the initial domain of continuous decision variables
See also	<code>KALIS_DEFAULT_CONTINUOUS_LB</code> <code>getub</code>

KALIS_DEFAULT_PRECISION_VALUE

Description	Default precision for continuous variables
Type	Double, read/write
Values	+1e-6 default value
Notes	Default precision for continuous variables
See also	setprecision

KALIS_DEFAULT_PRECISION_RELATIVITY

Description	Precision relativity for continuous variables
Type	Boolean, read/write
Values	false default value
Notes	Precision relativity for continuous variables
See also	setprecision

KALIS_DEFAULT_SCHEDULE_HORIZ_MIN

Description	Default minimal time horizon for schedules
Type	Integer, read/write
Values	0 default value
Notes	Default minimal time horizon for schedules
See also	KALIS_DEFAULT_SCHEDULE_HORIZ_MAX

KALIS_DEFAULT_SCHEDULE_HORIZ_MAX

Description	Default maximal time horizon for schedules
Type	Integer, read/write
Values	10000 default value
Notes	Default maximal time horizon for schedules
See also	KALIS_DEFAULT_SCHEDULE_HORIZ_MIN

KALIS_DICHOTOMIC_OBJ_SEARCH

Description	Uses a dichotomic strategy to find the optimal objective value
Type	Boolean, read/write
Values	false default value
Notes	Uses a dichotomic strategy to find the optimal objective value. By default the optimization algorithm imposes an amelioration constraint stating that the objective value of the solution to be found must be better than the objective value of the best solution already found. With a dichotomic optimization strategy, this constraint is replaced by a binary search tree over the objective variable domain.
Affects routines	cp_minimize cp_maximize

KALIS_USE_3B_CONSISTENCY

Description	Activates 3B consistency for continuous variables
Type	Boolean, read/write
Values	true default value
Notes	Activates 3B consistency for continuous variables
Affects routines	cp_propagate

KALIS_VERBOSE_LEVEL

Description	Verbosity level of Xpress Kalis
Type	Integer, read/write
Values	0 all output disabled 1 enable summary output printing 2 more detailed output 3 debug output
Notes	Verbosity level of Xpress Kalis

KALIS_AUTO_RELAX

Description	Indicates whether to use linear relaxation automatically
Type	Boolean, read/write

Values	false default value
Notes	Enables automatic problem relaxation.
See also	cp_propagate cp_post

KALIS_THREADS

Description	Number of threads
Type	Integer, read/write
Values	1
Notes	Sets the number of threads to be used during the search process. This parameter MUST be set before any variable, task or resource is declared/created.
Affects routines	cp_find_next_sol cp_minimize cp_maximize cp_schedule

KALIS_NARY_OBJ_SEARCH

Description	Uses an n-ary search strategy to find the optimal objective value
Type	Boolean, read/write
Values	false default value
Notes	Uses an n-ary search strategy to find the optimal objective value. By default the optimization algorithm imposes an amelioration constraint stating that the objective value of the solution to be found must be better than the objective value of the best solution already found. With an n-ary optimization strategy, this constraint is replaced by n search subtrees over the objective variable domain.
Affects routines	cp_minimize cp_maximize

KALIS_SCHEDULE_ENABLE_SHAVING

Description	Enable shaving during initial step of schedule search
Type	Boolean, read/write
Values	true default value
Notes	Enable shaving during initial step of schedule search. By default the optimization algorithm shaves the lower bound (for minimization) or upper bound (for maximization) of the objective variable.
Affects routines	cp_schedule

CHAPTER 9

Subroutines

9.1 Constraints

<code>abs</code>	Absolute value constraint	p. 81
<code>all_different</code>	All different constraint	p. 83
<code>and</code>	Conjunction composite constraint	p. 69
<code>cplintr</code>	Linear constraints	p. 67
<code>cpnlintr</code>	Non-linear constraints	p. 75
<code>cumulative</code>	Cumulative constraint	p. 101
<code>cycle</code>	Cycle constraint	p. 85
<code>disjunctive</code>	Disjunctive constraint	p. 103
<code>distance</code>	Distance constraint	p. 79
<code>distribute</code>	Distribute constraint with fixed bounds	p. 99
<code>element</code>	Element constraint	p. 94
<code>equiv</code>	Equivalence composite constraint	p. 71
<code>exp</code>	Exponential of a non-linear expression	p. 78
<code>generic_binary_constraint</code>	Generic Binary constraint	p. 96
<code>generic_nary_constraint</code>	Generic nary constraint	p. 98
<code>implies</code>	Implication composite constraint	p. 73
<code>ln</code>	Natural logarithm of a non-linear expression	p. 77
<code>maximum_minimum</code>	Maximum/minimum constraint	p. 90
<code>occurrence</code>	Occurrence constraint	p. 92
<code>or</code>	Disjunction composite constraint	p. 70
<code>producer_consumer</code>	Producer Consumer constraint	p. 105

cplnctr

Purpose

Linear constraints are arithmetic constraints over decision variables. Linear constraints are defined with linear expressions combined by arithmetic operators such as $=$, $\neq$, $\leq$, and $\geq$. Objects of type `cplnexp` are linear expressions of decision variables. Typically, these objects result as intermediate objects in the definition of linear constraints.

Linear expressions of decision variables are obtained by applying the operators $+$, $-$ and $*$ to decision variables and integer values. The standard priority rules used by Mosel apply to the evaluation order of the arithmetic operators. Brackets may be used in the definition of linear expressions to change this order. The following are examples of valid linear expressions:

```
declarations
  A, B: integer
  x, y: cpvar
  z: array(1..10) of cpfloatvar
end-declarations
A*x*B
-x + 2*A*y
B*(x-y)
(abs(A) + ceil(2.9) + round(-3.4)) * x
sum(i in 1..10) z(i)
```

The following definitions are not valid for objects of type `cplnexp`:

```
x*y
ln(2) * x
y/3
```

Synopsis

```
x op I with x:cpvar and I:integer and op one of =, ≥, ≤, <>
x op y with x,y:cpvar and op one of =, ≥, ≤, <>
z op R with z:cplnexp and R:real and op one of =, ≥, ≤, <>
z op x with z:cplnexp and x:cpvar and op one of =, ≥, ≤, <>
z op w with z,w:cplnexp and op one of =, ≥, ≤, <>
I op x with I:integer and x:cpvar and op one of =, ≥, ≤
R op z with R:real and z:cplnexp and op one of =, ≥, ≤, <>
x op z with x:cpvar and z:cplnexp and op one of =, ≥, ≤, <>
I <> z with I:integer and z:cplnexp
x op R with x:cpfloatvar and R:real and op one of ≥, ≤
```

Return value

A linear constraint combining the two arguments

Example

The following example shows how to state different kinds of linear constraints:

```
model "Linear constraints"
uses "kalis"

declarations
  N = 3
  R = 1..N
  x: array(R) of cpvar
  c1: cpctr
end-declarations

forall(i in R) do
  0 <= x(i); x(i) <= 50
```

```
        setname(x(i), "x"+i)
    end-do

! Automated post+propagation
x(1) >= x(3) + 5
writeln(x(1), " ", x(3))

! Named constraint (explicit post)
c1:= x(3) >= x(2) + 10

if cp_post(c1) then
    writeln("With constraint c1: ", x)
else exit(1)
end-if

! Stop automated propagation
setparam("kalis_auto_propagate", false)

! Automated post w/o propagation
x(2) >= 10
writeln("new bound for x2: ", x)
if cp_propagate then
    writeln("after propagation: ", x)
end-if

! Minimize the value of x(1)
if cp_minimize(x(1)) then
    write("Solution: ")
    forall(i in R) write(getsol(x(i)), " ")
    writeln
end-if
end-model
```

Related topics

[getval](#)

and

Purpose

This composite constraint states a conjunction between two constraints C1 and C2 ($C1 \wedge C2$). The satisfaction of the resulting constraint is given by the following truth table:

C1	C2	C1 and C2

false	false	false
false	true	false
true	false	false
true	true	true

Synopsis

C1 and C2

Arguments

- C1 the left member constraint of the conjunction
- C2 the right member constraint of the conjunction

Return value

A conjunction constraint over C1 and C2

Example

The following example shows how to use the conjunction constraint

```
model "Logical constraints"
  uses "kalis"

  ! Default bounds for all variables
  setparam("KALIS_DEFAULT_LB", 0); setparam("KALIS_DEFAULT_UB", 1)

  declarations
    a,b: cpvar
  end-declarations

  setname(a,"a")
  setname(b,"b")

  writeln(a,b)
  (a >= 1) and (b >= 1) or (a <= 0) and (b >= 1)

  while (cp_find_next_sol)
    writeln("a:", getsol(a), " b:", getsol(b))

  end-model
```

Related topics

[implies](#) and [or](#)

or

Purpose

This composite constraint states a disjunction between two constraints C1 and C2 ($C1 \vee C2$). The satisfaction of the resulting constraint is given by the following truth table:

C1	C2	C1 or C2
false	false	false
false	true	true
true	false	true
true	true	true

Synopsis

C1 or C2

Arguments

- C1 the left member constraint of the disjunction
- C2 the right member constraint of the disjunction

Return value

A disjunction constraint over C1 and C2

Example

The following example shows how to use the disjunction constraint

```
model "Logical constraints"
  uses "kalis"

  ! Default bounds for all variables
  setparam("KALIS_DEFAULT_LB", 0); setparam("KALIS_DEFAULT_UB", 1)

  declarations
    a,b: cpvar
  end-declarations

  setname(a,"a")
  setname(b,"b")

  writeln(a,b)
  (a >= 1) and (b >= 1) or (a <= 0) and (b >= 1)

  while (cp_find_next_sol)
    writeln("a:", getsol(a), " b:", getsol(b))

  end-model
```

Related topics

[implies and or](#)

equiv

Purpose

This composite constraint states an equivalence between two constraints C1 and C2 ($C1 \Leftrightarrow C2$), which are not individually posted. The satisfaction of the resulting constraint is given by the following truth table:

C1	C2	C1 equiv C2
false	false	true
false	true	false
true	false	false
true	true	true

Synopsis

```
function equiv(C1:cpctr,C2:cpctr) : cpctr
```

Arguments

C1 the left member constraint of the equivalence
C2 the right member constraint of the equivalence

Return value

An equivalence constraint over C1 and C2

Example

The following example shows how to use the equivalence constraint

```
model "Logical constraints"
  uses "kalis"

  ! Default bounds for all variables
  setparam("KALIS_DEFAULT_LB", 0); setparam("KALIS_DEFAULT_UB", 5)

  declarations
    A, B, C, D, E: cpctr
    x,y,z,b: cpvar
  end-declarations

  setname(b, "b"); setname(x, "x")
  setname(y, "y"); setname(z, "z")
  b <= 1

  ! Definition of constraints (without posting)
  A:= x >= y + 5 + z + b
  B:= b = 1
  C:= all_different({x, y, z})    ! Cannot be used with 'equiv'
  D:= y <= z
  E:= distance(y,z) <= 2

  writeln("Original domains: ", b, " ", x, " ", y, " ", z)

  ! If x, y and z are all different then b equals 1 (C=>B),
  ! if b equals 1 then x, y and z are all different (B=>C).
  implies(C, B)
  implies(B, C)

  ! E<=>B: b equals 1 if and only if |y-z| <= 2
```



```
equiv(E, B)

! A<=>D: x >= y + 5 + z + b if and only if y <= z
equiv(A, D)

writeln("With constraints: ", b, " ", x, " ", y, " ", z)

while (cp_find_next_sol)
    writeln("Solution: ", b, " ", x, " ", y, " ", z)

end-model
```

Related topics

[implies and or](#)

implies

Purpose

This composite constraint states an implication between two constraints C1 and C2 ($C1 \Rightarrow C2$), which are not individually posted. The satisfaction of the resulting constraint is given by the following truth table:

C1	C2	C1 implies C2
false	false	true
false	true	true
true	false	false
true	true	true

Synopsis

```
function implies(C1:cpctr, C2:cpctr) : cpctr
```

Arguments

C1 the left member constraint of the implication
C2 the right member constraint of the implication

Return value

An implication constraint over C1 and C2

Example

The following example shows how to use the implication constraint

```
model "Logical constraints"
  uses "kalis"

  ! Default bounds for all variables
  setparam("KALIS_DEFAULT_LB", 0); setparam("KALIS_DEFAULT_UB", 5)

  declarations
    A, B, C, D, E: cpctr
    x,y,z,b: cpvar
  end-declarations

  setname(b, "b"); setname(x, "x")
  setname(y, "y"); setname(z, "z")
  b <= 1

  ! Definition of constraints (without posting)
  A:= x >= y + 5 + z + b
  B:= b = 1
  C:= all_different({x, y, z})    ! Cannot be used with 'equiv'
  D:= y <= z
  E:= distance(y,z) <= 2

  writeln("Original domains: ", b, " ", x, " ", y, " ", z)

  ! If x, y and z are all different then b equals 1 (C=>B),
  ! if b equals 1 then x, y and z are all different (B=>C).
  implies(C, B)
  implies(B, C)

  ! E<=>B: b equals 1 if and only if |y-z| <= 2
```

```
equiv(E, B)

! A<=>D: x >= y + 5 + z + b if and only if y <= z
equiv(A, D)

writeln("With constraints: ", b, " ", x, " ", y, " ", z)

while (cp_find_next_sol)
    writeln("Solution: ", b, " ", x, " ", y, " ", z)

end-model
```

Related topics

[equiv and or](#)

cpnlinctr

Purpose

Non-linear constraints are arithmetic constraints over decision variables. Non-linear constraints are defined with non-linear expressions combined by arithmetic operators such as $=$, $\leq$, and $\geq$. Non-linear expressions of decision variables have the type `cpnlinexp`. Typically, these objects result as intermediate objects in the definition of non-linear constraints.

Non-linear expressions of decision variables are obtained by applying the operators $+$, $-$, $*$, div , or functions like `ln`, `exp`, `abs` to decision variables and numeric values. The standard priority rules used by Mosel apply to the evaluation order of the arithmetic operators. Brackets may be used in the definition of non-linear expressions to change this order. The following are examples of valid non-linear expressions:

```
declarations
  y: cpvar
  x1,x2,x3,x4,x5: cpfloatvar
end-declarations
0.4*x3*x5 + 0.1*x3*x4 >= 10.7
x1^2 = y
exp(x3-x4) 2
ln(x5) = 1
(x1*x2^2) div (x4-x3) = 3
```

Synopsis

```
X op v with X:cpnlinexp and v:cpfloatvar and op one of =, ≥, ≤
X op r with X:cpnlinexp and r:real and op one of =, ≥, ≤
X op v with X:cpnlinexp and v:cpvar and op one of =, ≥, ≤
X op l with X:cpnlinexp and l:cpllinexp and op one of =, ≥, ≤
```

Return value

A non-linear constraint combining the two arguments

Example

The following example shows how to state different kinds of non-linear constraints:

```
model "Non-linear constraints"
  uses "kalis"

  parameters
    PREC = 1e-10
  end-parameters

  ! Setting default precision of continuous variables
  setparam("KALIS_DEFAULT_PRECISION_VALUE", PREC)

  declarations
    ISET = 1..8
    x: array(ISET) of cpfloatvar
  end-declarations

  ! Setting variable names
  forall(i in ISET) x(i).name:= "x"+i

  ! Setting variable bounds
  forall(i in ISET) do
    x(i) >= -100; x(i) <= 100
  end-do
```

```
! Defining and posting non-linear constraints
x(1) + x(2)*(x(1)+x(3)) + x(4)*(x(3)+x(5)) + x(6)*(x(5)+x(7)) -
  (x(8)*((1/8)-x(7))) = 0
x(2) + x(3)*(x(1)+x(5)) + x(4)*(x(2)+x(6)) + x(5)*x(7) -
  (x(8)*((2/8)-x(6))) = 0
x(3)*(1 + x(6)) + x(4)*(x(1)+x(7)) + x(2)*x(5) -
  (x(8)*((3/8)-x(5))) = 0
x(4) + x(1)*x(5) + x(2)*x(6) + x(3)*x(7) - (x(8)*((4/8)-x(4))) = 0
x(5) + x(1)*x(6) + x(2)*x(7) - (x(8)*((5/8)-x(3))) = 0
x(6) + x(1)*x(7) - (x(8)*((6/8)-x(2))) = 0
x(7) - (x(8)*((7/8)-x(1))) = 0
sum(i in ISET) x(i) = -1

! Set the enumeration strategy
cp_set_branching(split_domain(KALIS_WIDEST_DOMAIN, KALIS_MIDDLE_VALUE,
                             x, true, 0))

! Find one solution
if (cp_find_next_sol) then
  writeln("Solution number 1" )
  cp_show_sol
  cp_show_stats
end-if

end-model
```

Related topics

[getval](#)

ln

Purpose

Natural logarithm of a non-linear expression

Synopsis

```
function ln(x:cpnlinexp) : cpnlinexp  
function ln(x:cpllinexp) : cpnlinexp  
function ln(x:cpfloatvar) : cpnlinexp  
function ln(x:cpvar) : cpnlinexp
```

Argument

x Non-linear expression

Return value

A natural logarithm constraint over a non-linear expression x: $y = \ln x$

Related topics

[exp](#) [abs](#)

exp

Purpose

Exponential of a non-linear expression

Synopsis

```
function exp(x:cpnlinexp) :  cpnlinexp  
function exp(x:cpnlinexp) :  cpnlinexp  
function exp(x:cpfloatvar) :  cpnlinexp  
function exp(x:cpvar) :  cpnlinexp
```

Argument

x Non-linear expression

Return value

An exponential constraint over a non-linear expression x: $y = e^x$

Related topics

[ln](#) [abs](#)

distance

Purpose

This constraint specifies the distance between two variables.

Synopsis

```
function distance(x:cpvar, y:cpvar) : 0cpabs
```

Argument

`x,y` finite domain variables

Return value

An absolute value constraint over `x` and `y` corresponding to the expressed distance constraint:
 $|x - y|$

Example

The following example shows how to use the distance constraint

```
model "Distance constraints"
uses "kalis"

setparam("KALIS_DEFAULT_LB", -50); setparam("KALIS_DEFAULT_UB", 50)

declarations
  x, y, z: cpvar
  Dist: cpctr
end-declarations

setname(x,"x")
setname(y,"y")
setname(z,"z")

abs(x) = y
writeln("Absolute value of x: ", x,y,z)

y >= 20
writeln("Bounding y by 20: ", x,y,z)

! abs(y-z) <= 3
! Equivalent version of this constraint:
distance(y,z) <= 3

writeln("Max distance betw. y and z: ", x,y,z)

Dist:= distance(x,z) = 5

if(cp_post(Dist)) then
  writeln("Distance between x and z: ", x,y,z)
else
  writeln("Problem is infeasible")
end-if

cp_show_prob

end-model
```


Related topics
[abs](#)

abs

Purpose

This constraint states that a variable y equals the absolute value of another variable x

Synopsis

```
abs(x) = y with x,y:cpvar
abs(x) op I with x:cpvar and I:integer and op one of =, ≥, ≤, <>
x = abs(y) with x,y:cpvar
I op abs(x) with I:integer and x:cpvar and op one of =, ≥, ≤, <>
```

Arguments

x	absolute value variable (of type cpvar, cpfloatvar, cpllinexp, cpnlinexp)
y	value variable (of the same type as x)
I	integer value

Return value

An absolute value constraint over the arguments

Example

The following example shows how to use the absolute value constraint

```
model "Distance constraints"
uses "kalis"

setparam("KALIS_DEFAULT_LB", -50); setparam("KALIS_DEFAULT_UB", 50)

declarations
  x, y, z: cpvar
  Dist: cpctr
end-declarations

setname(x,"x")
setname(y,"y")
setname(z,"z")

abs(x) = y
writeln("Absolute value of x: ", x,y,z)

y >= 20
writeln("Bounding y by 20: ", x,y,z)

! abs(y-z) <= 3
! Equivalent version of this constraint:
distance(y,z) <= 3

writeln("Max distance betw. y and z: ", x,y,z)

Dist:= distance(x,z) = 5

if(cp_post(Dist)) then
  writeln("Distance between x and z: ", x,y,z)
else
  writeln("Problem is infeasible")
end-if
```

`cp_show_prob`

`end-model`

Related topics

`distance`

all_different

Purpose

The all_different constraint states that all variables in this constraint must be pairwise different:

Synopsis

```
function all_different(vars:set of cpvar, propagation: integer) : cpctr
function all_different(vars:set of cpvar) : cpctr
function all_different(vars:array(range) of cpvar, propagation:integer) : cpctr
function all_different(vars:array(range) of cpvar) : cpctr
function all_different(vars:cpvarlist, propagation:integer) : cpctr
function all_different(vars:cpvarlist) : cpctr
```

Arguments

vars	the list of variables
propagation	Propagation type selected using a constant <code>KALIS_FORWARD_CHECKING</code> , <code>KALIS_GEN_ARC_CONSISTENCY</code>

Return value

An all different constraint over vars

Example

Consider the following simple problem where one must determine the arrival positions of a set of six runners: Dominique, Ignace, Naren, Olivier, Philippe and Pascal. The constraints are the following:

1. All runners have different positions
2. Olivier is not last
3. Dominique, Pascal, and Ignace are before Naren and Olivier
4. Dominique is better than third
5. Philippe is among the first four
6. Ignace is neither second nor third
7. Pascal is three places higher than Naren
8. Neither Ignace nor Dominique are in fourth position

This problem can be modeled as follows. We create one variable per runner, whose value will be his arrival position. The initial domains of the variables will be the available arrival positions (1-6). Since we admit there are no ties, the positions are all pairwise different.

```
model "all_different example"
uses "kalis"

declarations
  PEOPLE = {"Sebastian", "Frederic", "Jan-Georg",
            "Krzysztof", "Maarten", "Luca"} ! Set of speakers
  x: array(PEOPLE) of cpvar                ! Time slot per person
end-declarations

3 <= x("Sebastian") ; x("Sebastian") <= 6
3 <= x("Frederic")   ; x("Frederic")   <= 4
2 <= x("Jan-Georg")  ; x("Jan-Georg")  <= 5
2 <= x("Krzysztof")  ; x("Krzysztof")  <= 4
3 <= x("Maarten")    ; x("Maarten")    <= 4
1 <= x("Luca")       ; x("Luca")       <= 6
```

```
! A different time slot for every person
all_different(x)

! Solve the problem
if not(cp_find_next_sol) then
    writeln("Problem is infeasible")
    exit(1)
end-if

! Solution printout
writeln(x)

end-model
```

Related topics

[distribute KALIS_GEN_ARC_CONSISTENCY KALIS_FORWARD_CHECKING](#)

cycle

Purpose

The cycle constraint ensures that the graph implicitly represented by a set of variables (= nodes) and their domains (= possible successors of a node) contains no sub-tours, that is, tours visiting only a subset of the nodes. The constraint can take an optional second set of variables Preds, representing the inverse relation of the Succ function and ensure the following equivalences: $succ_i = j \Leftrightarrow pred_j = i$ for all i and j . Another optional parameter of the cycle constraint allows to take into account an accumulated quantity along the tour such as distance, time or weight. More formally, it ensures the following constraint: $quantity = \sum_{i,j} distmatrix_{ij}$ for all arcs $i \rightarrow j$ belonging to the tour.

Synopsis

```
function cycle(succ:array of cpvar) : cpctr
function cycle(succ:array of cpvar, pred:array of cpvar) : cpctr
function cycle(succ:array of cpvar, dist:cpvar, distmatrix:array(range,range) of
    integer) : cpctr
function cycle(succ:array of cpvar, pred:array of cpvar, dist:cpvar,
    distmatrix:array(range,range) of integer) : cpctr
```

Arguments

succ	the list of successors variables
pred	the list of predecessors variables
dist	the accumulated quantity variable
distmatrix	a (nodes × nodes) matrix of integers representing the quantity to add to the accumulated quantity variable when an edge (i,j) belongs to the tour.

Return value

A cycle constraint

Example

To illustrate the cycle constraint we show an implementation of the Traveling Salesman Problem (TSP). The objective of the Traveling Salesman Problem (TSP) is to find the shortest tour through a given set of cities that visits each city exactly once (a Hamiltonian tour). More formally, given a set of n points and a distance between every pair of points, a solution to the TSP is a path of N edges, with identical first and last vertices, containing all n points and with minimal total length. This problem can be modeled as follows: a solution is represented by a function Succ associating with each node its immediate successor. We use an array of N variables 'succ(i)' (one for each city $i \in \{0, \dots, N-1\}$) to represent the next city visited after city number i where the domain of the variables succ(i) is set to $\{0, \dots, N-1\} - \{i\}$.

```
model "TSP"
uses "kalis"

parameters
    S = 14 ! Number of cities to visit
end-parameters

declarations
    TC : array(0..3*S) of integer
end-declarations

! TSP DATA
TC :: [
    1 , 1647,  9610,
```

```
2 , 1647, 9444,  
3 , 2009, 9254,  
4 , 2239, 9337,  
5 , 2523, 9724,  
6 , 2200, 9605,  
7 , 2047, 9702,  
8 , 1720, 9629,  
9 , 1630, 9738,  
10, 1405, 9812,  
11, 1653, 9738,  
12, 2152, 9559,  
13, 1941, 9713,  
14, 2009, 9455]
```

```
forward procedure print_solution
```

```
setparam("KALIS_DEFAULT_LB", 0)  
setparam("KALIS_DEFAULT_UB", S-1)
```

```
declarations  
  CITIES = 0..S-1                ! Set of cities  
  succ: array(CITIES) of cpvar    ! Array of successor variables  
  prev: array(CITIES) of cpvar    ! Array of predecessor variables  
end-declarations
```

```
setparam("KALIS_DEFAULT_UB", 10000)
```

```
declarations  
  dist_matrix: array(CITIES,CITIES) of integer ! Distance matrix  
  totaldist: cpvar                          ! Total distance of the tour  
  succpred: cpvarlist                       ! Variable list for branching  
end-declarations
```

```
! Setting the variable names  
forall(p in CITIES) do  
  setname(succ(p), "succ("+p+")")  
  setname(prev(p), "prev("+p+")")  
end-do
```

```
! Add sucesors and predecessors to succpred list for branching  
forall(p in CITIES) succpred += succ(p)  
forall(p in CITIES) succpred += prev(p)
```

```
! Build the distance matrix  
forall(p1,p2 in CITIES | p1<>p2)  
  dist_matrix(p1,p2) := round(sqrt((TC(3*p2+1) - TC(3*p1+1)) *  
    (TC(3*p2+1) - TC(3*p1+1)) + (TC(3*p2+2) - TC(3*p1+2)) *  
    (TC(3*p2+2) - TC(3*p1+2))))
```

```
! Set the name of the distance variable  
setname(totaldist, "total_distance")
```

```
! Posting the cycle constraint  
cycle(succ, prev, totaldist, dist_matrix)
```

```

! Print all solutions found
cp_set_solution_callback("print_solution")

! Set the branching strategy
cp_set_branching(assign_and_forbid("bestregret", "bestneighbor",
                                   succpred))
setparam("KALIS_MAX_COMPUTATION_TIME", 5)

! Find the optimal tour
if (cp_minimize(totaldist)) then
  if getparam("KALIS_SEARCH_LIMIT")=KALIS_SLIM_BY_TIME then
    writeln("Search time limit reached")
  else
    writeln("Done!")
  end-if
end-if

!-----
! **** Solution printing ****
procedure print_solution
  writeln("TOUR LENGTH = ", getsol(totaldist))

  thispos:=getsol(succ(0))
  nextpos:=getsol(succ(thispos))
  write("  Tour: ", thispos)
  while (nextpos <> getsol(succ(0))) do
    write(" -> ", nextpos)
    thispos:=nextpos
    nextpos:=getsol(succ(thispos))
  end-do
  writeln
end-procedure

!-----
! **** Variable choice ****
function bestregret(Vars: cpvarlist): integer

! Get the number of elements of "Vars"
listsize:= getsize(Vars)
minindex := 0
mindist := 0

! Set on uninstantiated variables
forall(i in 1..listsize) do
  if not is_fixed(getvar(Vars,i)) then
    if (i <= S) then
      bestn := getlb(getvar(Vars,i))
      v:=bestn
      mval:=dist_matrix(i-1,v)
      while (v < getub(getvar(Vars,i))) do
        v:=getnext(getvar(Vars,i),v)
        if dist_matrix(i-1,v)<=mval then
          mval:=dist_matrix(i-1,v)
          bestn:=v
        end-if
      end-while
    end-if
  end-if
end-forall

```



```

        end-if
    end-do
    sbestn := getlb(getvar(Vars,i))
    mval2:= 10000000
    v:=sbestn
    if (dist_matrix(i-1,v)<=mval2 and v <> bestn) then
        mval2:=dist_matrix(i-1,v)
        sbestn:=v
    end-if
    while (v < getub(getvar(Vars,i))) do
        v:=getnext(getvar(Vars,i),v)
        if (dist_matrix(i-1,v)<=mval2 and v <> bestn) then
            mval2:=dist_matrix(i-1,v)
            sbestn:=v
        end-if
    end-do

else

    bestn := getlb(getvar(Vars,i))
    v:=bestn
    mval:=dist_matrix(v,i-S-1)
    while (v < getub(getvar(Vars,i))) do
        v:=getnext(getvar(Vars,i),v)
        if dist_matrix(v,i-S-1)<=mval then
            mval:=dist_matrix(v,i-S-1)
            bestn:=v
        end-if
    end-do
    sbestn := getlb(getvar(Vars,i))
    mval2:= 10000000
    v:=sbestn
    if (dist_matrix(v,i-S-1)<=mval2 and v <> bestn) then
        mval2:=dist_matrix(v,i-S-1)
        sbestn:=v
    end-if
    while (v < getub(getvar(Vars,i))) do
        v:=getnext(getvar(Vars,i),v)
        if (dist_matrix(v,i-S-1)<=mval2 and v <> bestn) then
            mval2:=dist_matrix(v,i-S-1)
            sbestn:=v
        end-if
    end-do
end-if

dsize := getsize(getvar(Vars,i))

rank := integer(10000/ dsize +(mval2 - mval))
if (mindist<= rank) then
    mindist := rank
    minindex := i
end-if

end-if
end-do

```

```
        returned := minindex

end-function

!-----
! **** Value choice: choose value resulting in smallest distance
function bestneighbor(x: cpvar): integer

    issucc := false
    idx := -1
    forall (i in CITIES)
        if (is_same(succ(i),x)) then
            idx:= i
            issucc := true
        end-if
    forall (i in CITIES)
        if (is_same(prev(i),x)) then
            idx:= i
        end-if

    if issucc then
        returned:= getlb(x)
        v:=getlb(x)
        mval:=dist_matrix(idx,v)
        while (v < getub(x)) do
            v:=getnext(x,v)
            if dist_matrix(idx,v)<=mval then
                mval:=dist_matrix(idx,v)
                returned:=v
            end-if
        end-do
    else
        returned:= getlb(x)
        v:=getlb(x)
        mval:=dist_matrix(v,idx)
        while (v < getub(x)) do
            v:=getnext(x,v)
            if dist_matrix(v,idx)<=mval then
                mval:=dist_matrix(v,idx)
                returned:=v
            end-if
        end-do
    end-if

end-function

end-model
```

Related topics

[distribute all_different](#)

maximum_minimum

Purpose

The maximum (resp) minimum constraint states that a variable *z* is the maximum (resp) minimum of a list of variables *vars*

Synopsis

```
minimum(vars) = z or maximum(vars) = z  
z = minimum(vars) or z = maximum(vars)
```

Arguments

vars list of decision variables: {set of cpvar | array(range) of cpvar | cpvarlist}
z decision variable for the maximum/minimum: cpvar

Return value

A maximum/minimum constraint over *z* and *vars*

Example

The following example shows how to use the maximum/minimum constraints:

```
model "Min and Max"  
uses "kalis"  
  
declarations  
  R = 1..5  
  x: array(R) of cpvar  
  v, w, y: cpvar  
  L: cpvarlist  
  MaxCtr: cpctr  
end-declarations  
  
setname(v,"v")  
setname(w,"w")  
setname(y,"y")  
  
forall(i in R) do  
  setname(x(i),"x"+i+"")  
  setdomain(x(i), 0, 2*i + round(5*random + 0.5))  
end-do  
  
writeln("Initial domains:\n ", x, " ", v)  
  
! Minimum constraint with automated posting  
v = minimum(x)  
writeln("With minimum constraint:\n ", x, " ", v)  
  
x(1) = 2  
writeln("Fixing x(1) to 2: ", v)  
  
! Maximum constraint with explicit posting  
MaxCtr:= w = maximum({x(2), x(3), x(5)})  
if cp_post(MaxCtr) then  
  writeln("With maximum constraint:\n ", x, " ", w)  
else exit(1)  
end-if
```

```
w <= 7
writeln("Bounding w by 7:\n ", x, " ", w)

! Maximum constraint on list of variables
L += x(2); L += x(3); L += x(4)
y = maximum(L)
writeln("With 2nd maximum constraint:\n ", x, " ", y)

if (cp_find_next_sol) then
  writeln("A solution:\n ", x, " ", v, " ", w, " ", y)
end-if
end-model
```

occurrence

Purpose

Constrains the number of occurrences of a specific value or a set of values among a list/set of decision variables

Synopsis

occurrence(value,vars) *op* target with target:integer and *op* one of =, ≥, ≤
 target *op* occurrence(value,vars) with target:integer and *op* one of =, ≥, ≤
 occurrence(value,vars) *op* occvar with occvar:cpvar and *op* one of =, ≥, ≤
 occvar *op* occurrence(value,vars) with occvar:cpvar and *op* one of =, ≥, ≤
 occurrence(varset:set of cpvar, values:set of integer, minocc:integer, maxocc:integer)

Arguments

target	the (fixed) target occurrence count for the specified value
occvar	the variable occurrence count for the specified value
value	the integer value whose occurrences in the variable list should be constrained
vars	the list of decision variables: {set of cpvar array of cpvar cpvarlist}
varset	a set of decision variables
values	a set of values
minocc	minimum number of occurrences
maxocc	maximum number of occurrences

Return value

An occurrence constraint over the given arguments

Example

The following example shows how to use the occurrence constraint:

```
model "Cardinality"
  uses "kalis"

  setparam("KALIS_DEFAULT_LB", 0)

  declarations
    R = 1..6
    S = 1..10
    x: array(R) of cpvar
    y: array(S) of cpvar
    a,b,c: cpvar
    Card: cpctr
    Vlist: cpvarlist
  end-declarations

  forall(i in R) setname(x(i), "x"+i)
  forall(i in 1..3) x(i) = 1
  forall(i in 4..6) x(i) <= 10
  setname(c, "c")
  c <= 15

  writeln("Initial domains:\n ", x, " ", c)

  ! Explicit posting of an occurrence constraint
  Card:= occurrence(1, x) = c
```

```
if cp_post(Card) then
  writeln("With occurrence constraint:\n ", x, " ", c)
else exit(1)
end-if

c = 6
writeln("Fixing occurrence to 6:\n ", x, " ", c)

forall(i in S) do
  setname(y(i), "y"+i)
  i <= y(i); y(i) <= i*2
end-do
setname(a, "a"); setname(b,"b")

writeln("Initial domains:\n ", y, " ", a, " ", b)

! Occurrence constraint on an array of variables
occurrence(4, y) <= 2

! Occurrence constraint on a list of variables
Vlist += y(1); Vlist += y(2); Vlist += y(4)
occurrence(2, Vlist) = 0

! Occurrence constraint on a set of variables
occurrence(9, {y(6), y(7), y(9)}) >= 3

! Occurrences bounded by variables
occurrence(5, y) >= a
occurrence(8, {y(4), y(5), y(6), y(7), y(8)}) <= b
writeln("With all constraints:\n ", y, " ", a, " ", b)

if cp_find_next_sol then
  writeln("A solution:\n ", y, " ", a, " ", b)
end-if

end-model
```

Related topics

[distribute](#)

element

Purpose

This constraint states that a variable z is the x^{th} element of an ordered list of integer V , in its ternary form it states that z is the $[x,y]$ -th element of a matrix of integers M

Synopsis

```
element(x+I) = C
element(V,x{,I}) = C
element(V,x{,I}) = z with x,z cpvar and I integer
z = element(V,x{,I}) with x,z cpvar and I integer
element(M,x,y) = z with x,y,z cpvar
z = element(M,x,y) with x,y,z cpvar
```

Arguments

C	a constant integer value
z	the value variable
x	first index variable
y	second index variable
I	optional constant offset for index
V	a one-dimensional array of integer values
M	a matrix (two-dimensional array) of integers

Return value

An element constraint over z , x and y in the ternary form, over x and z in the binary form

Example

The following example shows how to use the element constraint:

```
model "Element"
uses "kalis"

declarations
  RY = 43..52
  RX = 1..2
  D: array(RY) of integer
  D2: array(RX,RY) of integer
  x,y,d_of_y,d_of_x_y: cpvar
end-declarations

D :: (43..52)[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

D2:: (1..2,43..52)[10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                  20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

setname(x, "x")
setname(y, "y")
setname(d_of_y, "d_of_y")
setname(d_of_x_y, "d_of_x_y")

writeln("Original domains: ", x, y, d_of_y, d_of_x_y)

element(D,y) = d_of_y
element(D2,x,y) = d_of_x_y
```

```
writeln("After propagation: ", x, y, d_of_y, d_of_x_y)

! Solve the problem
while (cp_find_next_sol) do
  nbSolutions += 1
  writeln("Solution ", nbSolutions, ": x:", getsol(x),
    " y:", getsol(y), " d_of_y:", getsol(d_of_y),
    " d_of_x_y:", getsol(d_of_x_y))
end-do
writeln("done!")

end-model
```


generic_binary_constraint

Purpose

This constraint can be used to propagate a user-defined constraint over two variables (its propagation is based on the AC2001 algorithm (cf. [BR01])).

Synopsis

```
function generic_binary_constraint(v1:cpvar,v2:cpvar, fctname:string) : cpctr
```

Arguments

v1	the first decision variable
v2	the second decision variable
fctname	name of the function specifying the user-defined constraint, such a function necessarily takes two cpvar as arguments and returns a Boolean.

Return value

A binary constraint over 'v1' and 'v2'

Example

The following example shows how to use the generic_binary_constraint constraint to solve the classical Euler Knight Tour problem:

```
model "Euler Knight Moves"
uses "kalis"

parameters
  S = 8                                ! No. of rows/columns
end-parameters

N:= S * S                              ! Total number of cells
setparam("KALIS_DEFAULT_LB", 0)
setparam("KALIS_DEFAULT_UB", N-1)

declarations
  PATH = 1..N                          ! Cells on the board
  pos: array(PATH) of cpvar             ! Position p in tour
end-declarations

! Setting names of decision variables
forall(i in PATH) setname(pos(i), "Position"+i)

! Fix the start position
pos(1) = 0

! Each cell is visited once
all_different(pos, KALIS_GEN_ARC_CONSISTENCY)

! The knight's path obeys the chess rules for valid knight moves
forall(i in 1..N-1)
  generic_binary_constraint(pos(i), pos(i+1), "valid_knight_move")
generic_binary_constraint(pos(N), pos(1), "valid_knight_move")

! Setting enumeration parameters
cp_set_branching(probe_assign_var(KALIS_SMALLEST_MIN,
                                  KALIS_MAX_TO_MIN, pos, 14))
```

```
! Search for up to NBSOL solutions
solct:= 0
if not cp_find_next_sol then
  writeln("No solution")
else
  writeln(pos)
end-if

! **** Test whether the move from a to b is admissible ****
function valid_knight_move(a:integer, b:integer): boolean
  declarations
    xa,ya,xb,yb,delta_x,delta_y: integer
  end-declarations
  xa := a div S
  ya := a mod S
  xb := b div S
  yb := b mod S
  delta_x := abs(xa-xb)
  delta_y := abs(ya-yb)
  returned := (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3)
end-function

end-model
```

generic_nary_constraint

Purpose

This constraint can be used to propagate a user-defined constraint over n variables (its propagation is based on the GAC2001 algorithm (cf. [BR01])).

Synopsis

```
function generic_nary_constraint(vars:array of cpvar, fctname:string,  
                                userparam:integer) : cpctr  
function generic_nary_constraint(vars:cpvarlist, fctname:string, userparam:integer) :  
    cpctr  
function generic_nary_constraint(vars:array of cpvar, fctname:string, propagation:  
    integer, userparam:integer) : cpctr  
function generic_nary_constraint(vars:cpvarlist, fctname:string, propagation:  
    integer, userparam:integer) : cpctr  
function generic_nary_constraint(vars:set of cpvar, fctname:string, userparam:integer)  
    : cpctr  
function generic_nary_constraint(vars:set of cpvar, fctname:string, propagation:  
    integer, userparam:integer) : cpctr
```

Arguments

vars	a set, array, or cpvarlist of decision variables
fctname	name of the function specifying the user-defined constraint, such a function necessarily takes a cpvarlist/cptuple and an integer (the value of userparam) as arguments and returns a Boolean.
userparam	a user parameter
propagation	the level of propagation to achieve

Return value

An n-ary constraint over a set of variables

distribute

Purpose

A distribute constraint also known as GCC (Global Cardinality Constraint) over a set of variables is defined by three arrays called values, lowerBound, and upperBound. The constraint is satisfied if, and only if, the number of variables of the given set, which are assigned to values[i], is greater than or equal to lowerBound[i], and less than or equal to upperBound[i] for all i, and if no variable of the given set is assigned to a value which does not belong to values. The constraint is equivalent, from a modelization point of view, to posting two instances of **occurrence** constraints for each value. But this is absolutely not equivalent from a propagation point of view: distribute acquires a far better propagation, using the Regin algorithm.

Synopsis

```
function distribute(vars:array of cpvar, values:set of integer, bound:array(integer)
    of integer) : cpctr
function distribute(vars:array of cpvar, values:set of integer,
    lowerBound:array(integer) of integer, upperBound:array(integer) of integer) :
    cpctr
function distribute(vars:set of cpvar,vals:set of integer, lowerBound:array(integer)
    of integer, upperBound:array(integer) of integer) : cpctr
function distribute(vars:cpvarlist, vals:set of integer, lowerBound:array(integer) of
    integer, upperBound:array(integer) of integer) : cpctr
```

Arguments

vars	Array/set/list of variables
values	Set of values represented by their names
lowerBound	Array of lower bounds on occurrences
upperBound	Array of upper bounds on occurrences
bound	Array of occurrence values

Return value

A distribute constraint

Example

A simple planning problem for personnel in a theater. Suppose that a movie theatre director has to decide in which location each of his employees should be posted. There are eight employees: Andrew, David, Jane, Jason, Leslie, Michael, Marilyn and Oliver. There are four locations: the ticket office, the first entrance, the second entrance and the coat check. These locations require three, two, two, and one person respectively. This constraint will be modeled by one distribute constraint:

```
model "Distribute example"
    uses "kalis"

    declarations
        PERS = {"David","Andrew","Leslie","Jason","Oliver","Michael",
            "Jane","Marilyn"}          ! Set of personnel
        LOC = 1..4                    ! Set of locations
        REQ: array(LOC) of integer    ! No. of pers. req. per loc.
        place: array(PERS) of cpvar   ! Workplace for each peson
    end-declarations

    ! Initialize data
    REQ:: [3, 2, 2, 1]
```

```
! Each variable has a lower bound of 1 (Ticket office) and
! an upper bound of 4 (Cloakroom)
forall(p in PERS) do
    setname(place(p), "workplace("+p+")")
    1 <= place(p); place(p) <= 4
end-do

! Creation of a resource constraint of for every location
forall(d in LOC) occurrence(d, place) = REQ(d)

! Elegant way to declare theses constraints,
! moreover achieving stronger pruning (using global
! cardinality constraint)
distribute(place, LOC, REQ)

! Solve the problem
if not(cp_find_next_sol) then
    writeln("Problem is infeasible")
    exit(1)
end-if

! Solution printout
writeln(place)

end-model
```

Related topics

[occurrence all_different](#)

cumulative

Purpose

This constraint states that the tasks requiring a resource do not exceed the resource capacity. The primary use of this constraint is to express resource constraints.

Synopsis

```
function cumulative(starts: array(integer) of cpvar, durations:array(integer) of
    cpvar, ends: array(integer) of cpvar, usages: array(integer) of cpvar, sizes:
    array(integer) of cpvar, C: integer) : cpctr
function cumulative(starts: array(integer) of cpvar, durations:array(integer) of
    cpvar, ends: array(integer) of cpvar, usages: array(integer) of cpvar, sizes:
    array(integer) of cpvar, C: array(integer) of integer) : cpctr
```

Arguments

<code>starts</code>	Array of variables representing the start times of the tasks
<code>ends</code>	Array of variables representing the completion times of the tasks
<code>durations</code>	Array of variables representing the durations of the tasks
<code>usages</code>	Array of variables representing the resource consumptions of the tasks
<code>sizes</code>	Array of variables representing the sizes of the tasks
<code>C</code>	integer representing the initial capacity of the resource (constant over time or capacity value for each time period)

Return value

A cumulative constraint ensuring that the maximal resource capacity is never exceeded. More formally the constraint ensures that

$starts_j + durations_j = ends_j$ for all j in Tasks

$usages_j \cdot durations_j = sizes_j$ for all j in Tasks

$\sum_{j \in \text{Tasks} | t \in [UB(start_j), LB(end_j)]} usages_j \leq C_t$ for all times t in the planning period

Example

The following example shows how to use the cumulative constraint to express resource constraints for five tasks using the same resource:

```
model "Cumulative scheduling"
uses "kalis"

declarations
    TASKS = 1..5
    obj : cpvar
    starts, ends, durations, usages, sizes : array(TASKS) of cpvar
end-declarations

C := 2                                ! Resource capacity
HORIZON := 10                        ! Time horizon

! Setting up the variables representing task properties
forall (t in TASKS) do
    starts(t).name:= "T"+t+".start"
    ends(t).name:= "T"+t+".end"
    durations(t).name:= "T"+t+".duration"
    sizes(t).name:= "T"+t+".size"
    usages(t).name:= "T"+t+".use"
```

```
0 <= starts(t); starts(t) <= HORIZON
0 <= ends(t); ends(t) <= HORIZON
t <= durations(t); durations(t) <= t+1
1 <= sizes(t); sizes(t) <= 100
1 <= usages(t); usages(t) <= 1
obj >= ends(t)
end-do

! Cumulative resource constraint
cumulative(starts, durations, ends, usages, sizes, C)

! Define the branching strategy
cp_set_branching(assign_var(KALIS_SMALLEST_MIN,KALIS_MIN_TO_MAX))

! Solve the problem
if cp_minimize(obj) then
  cp_show_sol
  write("Resource use profile: ")
  forall(t in TASKS, time in 0..HORIZON)
    if (starts(t).sol <= time) and (ends(t).sol > time) then
      rload(time) += usages(t).sol
    end-if
  forall(time in 0..HORIZON) write(rload(time))
  writeln
else
  writeln("No solution found")
end-if

end-model
```

Related topics

[disjunctive settle_disjunction](#) or

disjunctive

Purpose

This constraint states that the given tasks are not overlapping chronologically.

Synopsis

```
procedure disjunctive(starts: set of cvar, durations:array(cvar) of integer,
                     disj:set of cpctr, resource:integer)
procedure disjunctive(starts: array(integer) of cvar, durations:array(integer) of
                     cvar, ends: array(integer) of cvar)
```

Arguments

starts	Array of variables representing the start times of the tasks
durations	Array of integers representing the durations of the tasks
ends	Array of variables representing the completion times of the tasks
disj	Empty array that will be filled with the list of disjunctions that will be created by this constraint
resource	Resource flag (argument currently unused)

Return value

A disjunctive constraint ensuring that the tasks defined by 'starts' and 'durations' are not overlapping chronologically.

Example

The following example shows how to use the disjunctive constraint to express resource constraints in a small disjunctive scheduling problem:

```
model "Disjunctive scheduling with settle_disjunction"
uses "kalis"

declarations
  NBTASKS = 5
  TASKS = 1..NBTASKS                                ! Set of tasks
  DUR: array(TASKS) of integer                       ! Task durations
  DURs: array(set of cvar) of integer                 ! Durations
  DUE: array(TASKS) of integer                       ! Due dates
  WEIGHT: array(TASKS) of integer                    ! Weights of tasks
  start: array(TASKS) of cvar                        ! Start times
  tmp: array(TASKS) of cvar                          ! Aux. variable
  tardiness: array(TASKS) of cvar                    ! Tardiness
  twt: cvar                                           ! Objective variable
  zeroVar: cvar                                       ! 0-valued variable
  Strategy: array(range) of cpbranching              ! Branching strategy
  Disj: set of cpctr                                 ! Disjunctions
end-declarations

DUR :: [21,53,95,55,34]
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

! Setting up the decision variables
```



```
forall (t in TASKS) do
    start(t) >= 0
    setname(start(t), "Start("+t+")")
    DURs(start(t)):= DUR(t)
    tmp(t) = start(t) + DUR(t) - DUE(t)
    setname(tardiness(t), "Tard("+t+")")
    tardiness(t) = maximum({tmp(t), zeroVar})
end-do

twt = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
disjunctive(union(t in TASKS) {start(t)}, DURs, Disj, 1)

! Define the search strategy
Strategy(1):= settle_disjunction
Strategy(2):= split_domain(KALIS_LARGEST_MIN,KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)
setparam("KALIS_DICHOTOMIC_OBJ_SEARCH",true)

if not(cp_minimize(twt)) then
    writeln("Problem is inconsistent")
    exit(0)
end-if

forall (t in TASKS)
    writeln("[", getsol(start(t)), "==>",
            getsol(start(t)) + DUR(t), "]:\t ",
            getsol(tardiness(t)), "  (" , getsol(tmp(t)), ")")
writeln("Total weighted tardiness: ", getsol(twt))

end-model
```

Related topics

[or settle_disjunction](#)

producer_consumer

Purpose

A Producer Consumer Scheduling constraint. More formally the constraint ensures that:

$productions_j \cdot durations_j = prodsizes_j$ for all j in Tasks

$consumptions_j \cdot durations_j = consosizes_j$ for all j in Tasks

$\sum_{j \in R | t \in [UB(start_j) .. LB(end_j)]} (productions_j - consumptions_j) \leq C_t$ for all t in Times

Synopsis

```
function producer_consumer(starts:array(range) of cpvar, ends:array(range) of cpvar,
    durations:array(range) of cpvar, productions:array(range) of cpvar,
    prod_sizes:array(range) of cpvar, consumptions:array(range) of cpvar,
    conso_sizes:array(range) of cpvar, C:array(range) of integer) : cpctr
```

Arguments

starts	array of starting times
ends	array of ending times
durations	array of durations
productions	array of tasks' requirements
prod_sizes	array of tasks' productions
consumptions	array of tasks' provisions
conso_sizes	array of tasks' consumptions
C	initial resource capacity array indexed by time

Example

The following example shows how to use the producer_consumer constraint for the problem of planning the construction of a house (an example of resource-constrained project scheduling).

```
model "Cumulative Scheduling"
    uses "kalis"

    setparam("KALIS_DEFAULT_LB",0)
    setparam("KALIS_DEFAULT_UB",100)

    declarations
        ! Task indices
        Masonry = 1; Carpentry= 2; Roofing      = 3; Windows      = 4
        Facade  = 5; Garden   = 6; Plumbing     = 7; Ceiling      = 8
        Painting= 9; MovingIn =10; InitialPayment=11; SecondPayment=12
        BUILDTASKS = 1..10
        PAYMENTS = 11..12
        TASKS = BUILDTASKS+PAYMENTS
        TNames: array(TASKS) of string

    obj:cpvar
    starts : array(TASKS) of cpvar      ! Start times variables
    ends   : array(TASKS) of cpvar      ! Completion times
    durations: array(TASKS) of cpvar    ! Durations of tasks
    consos  : dynamic array(TASKS) of cpvar ! Res. consumptions
    sizes   : dynamic array(TASKS) of cpvar ! Consumption sizes
    prods   : dynamic array(TASKS) of cpvar ! Res. production
    sizep   : dynamic array(TASKS) of cpvar ! Production sizes
```

```

    Strategy : cpbranching                ! Branching strategy
end-declarations

TNAMES:: (1..12) ["Masonry", "Carpentry", "Roofing", "Windows",
                 "Facade", "Garden", "Plumbing", "Ceiling", "Painting",
                 "MovingIn", "InitialPayment", "SecondPayment"]

! Setting the names of the variables
forall(j in TASKS) do
    starts(j).name := TNAMES(j)+".start"
    ends(j).name := TNAMES(j)+".end"
    durations(j).name := TNAMES(j)+".duration"
end-do

! Creating consumption variables
forall(j in BUILDTASKS) do
    create(sizes(j))
    sizes(j).name := TNAMES(j)+".size"
    create(consos(j))
    consos(j).name := TNAMES(j)+".conso"
end-do

! Setting durations of building tasks
durations(Masonry) =7; durations(Carpentry)=3; durations(Roofing) =1
durations(Windows) =1; durations(Facade) =2; durations(Garden) =1
durations(Plumbing)=8; durations(Ceiling) =3; durations(Painting)=2
durations(MovingIn)=1

! Precedence constraints among building tasks
starts(Carpentry) >= ends(Masonry)
starts(Roofing) >= ends(Carpentry)
starts(Windows) >= ends(Roofing)
starts(Facade) >= ends(Roofing)
starts(Garden) >= ends(Roofing)
starts(Plumbing) >= ends(Masonry)
starts(Ceiling) >= ends(Masonry)
starts(Painting) >= ends(Ceiling)
starts(MovingIn) >= ends(Windows)
starts(MovingIn) >= ends(Facade)
starts(MovingIn) >= ends(Garden)
starts(MovingIn) >= ends(Painting)

! Setting task consumptions
consos(Masonry) = 7; consos(Carpentry) = 3; consos(Roofing) = 1
consos(Windows) = 1; consos(Facade) = 2; consos(Garden) = 1
consos(Plumbing) = 8; consos(Ceiling) = 3; consos(Painting) = 2
consos(MovingIn) = 1

! Production (amount) of payment tasks
forall(j in PAYMENTS) do
    create(prods(j))
    prods(j).name := TNAMES(j)+".prod"
    create(sizep(j))
    sizep(j).name := TNAMES(j)+".sizep"
end-do

```

```
! Payment data
prods(InitialPayment)    = 20; prods(SecondPayment)    = 9
durations(InitialPayment) = 1; durations(SecondPayment) = 1
starts(InitialPayment)   = 0; starts(SecondPayment)   = 15

! Objective: makespan of the schedule
obj = maximum({ ends(Masonry) , ends(Carpentry), ends(Roofing),
               ends(Windows), ends(Facade), ends(Garden), ends(Plumbing),
               ends(Ceiling), ends(Painting), ends(MovingIn)})

! Posting the producer_consumer constraint
producer_consumer(starts,ends,durations,prods,sizep,consos,sizes)

! Setting the search strategy
Strategy:= assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX, starts)
cp_set_branching(Strategy)

! Find the optimal solution
if cp_minimize(obj) then
    writeln("Minimum makespan: ", obj.sol)
    forall(j in BUILDTASKS)
        writeln(TNAMES(j), ": ", starts(j).sol, " - ", ends(j).sol)
    else
        writeln("No solution found")
    end-if
end-model
```

Related topics

[cumulative](#)

9.2 Constraint parameters

<code>getactivebranch</code>	Gets the active branch of a disjunction	p. 115
<code>getarity</code>	Returns the number of variables in the constraint	p. 109
<code>getpriority</code>	Returns the priority of a constraint	p. 110
<code>gettag</code>	Gets the tag of a constraint	p. 112
<code>setfirstbranch</code>	Sets the first branch of a disjunction to be activated	p. 114
<code>setpriority</code>	Sets the priority of a constraint	p. 111
<code>settag</code>	Sets the tag of a constraint	p. 113

getarity

Purpose

Returns the number of decision variables involved in the constraint; this function can be used, for instance, to design advanced search heuristics

Synopsis

```
function getarity(ctr:cpctr) : integer
```

Argument

`ctr` the constraint to explore

Return value

The number of variables involved in `ctr`

Example

The following example shows how to get the arity of a constraint

```
getarity(x + y = 3)
```

getpriority

Purpose

Returns the priority of a disjunction passed in argument; this feature is intended for advanced use in dynamically prioritizing the disjunction for solving (highest priority constraints are handled first)

Synopsis

```
function getpriority(ctr:cpctr) : integer
```

Argument

ctr The constraint to explore (disjunction only)

Return value

The associated priority

Example

The following example shows how to get the priority of a constraint

```
getpriority((x + y = 3) or (x + y = 1))
```

Related topics

[setpriority](#)

setpriority

Purpose

Sets the priority of a constraint passed in argument

Synopsis

```
procedure setpriority(ctr:cpctr, priority:integer)
```

Arguments

ctr	the constraint (disjunction only)
priority	a user-defined integer (the higher it will be, the quicker the disjunction will be activated)

Example

The following example shows how to set the priority of a constraint:

```
setpriority((x + y = 3) or (x + y = 1),2)
```

Related topics

[getpriority](#)

gettag

Purpose

Gets the tag (user-defined integer information) of a constraint

Synopsis

```
function gettag(ctr:cpctr) : integer
```

Argument

ctr the constraint

Return value

The tag that has been previously associated to it

Example

The following example shows how to get the tag of a constraint ctr

```
writeln(gettag(ctr))
```

Related topics

[settag](#)

settag

Purpose

Sets the tag (user-defined integer information) of a constraint

Synopsis

```
procedure settag(ctr:cpctr,tag:integer)
```

Arguments

ctr	the constraint
tag	the tag

Example

The following example shows how to set the tag of a constraint to '32'

```
settag(ctr,32)
```

Related topics

[gettag](#)

setfirstbranch

Purpose

This procedure enables the user to specify the first constraint of a disjunction to be activated; thus, it is possible to create dynamical heuristics for disjunctions.

Synopsis

```
procedure setfirstbranch(disj:cpctr, firstbranch:integer)
```

Arguments

disj	the involved constraint (only a disjunction)
firstbranch	0 to activate the first part and 1 else

Related topics

[disjunctive](#) or [getactivebranch](#)

getactivebranch

Purpose

This function enables the user to know which part of a disjunction is active; if the first constraint of the disjunction is satisfied, it returns zero, else it returns one.

Synopsis

```
function getactivebranch(disj:cpctr) : integer
```

Argument

`disj` the involved constraint (only a disjunction)

Return value

Zero if the first part of the disjunction is satisfied, one otherwise.

Example

The following code shows how to retrieve the active branch information from the disjunction `disj`:

```
b:=getactivebranch(disj)
```

Related topics

[disjunctive or setfirstbranch](#)

9.3 Variables

<code>contains</code>	Tests if a value is in the domain of a variable	p. 128
<code>cp_show_var</code>	Shows the current domain of the variable	p. 137
<code>getdegree</code>	Returns the degree of a variable	p. 123
<code>getlb</code>	Returns the current lower bound of a variable	p. 117
<code>getmiddle</code>	Returns the middle value of a variable	p. 119
<code>getnext</code>	Gets the next value in the domain of a finite domain variable	p. 126
<code>getprev</code>	Gets the previous value in the domain of a finite domain variable	p. 127
<code>getrand</code>	Returns a random value belonging to the domain of a variable	p. 125
<code>getsize</code>	Returns the cardinality of the variable domain	p. 120
<code>gettarget</code>	Returns the target value of a variable	p. 124
<code>getub</code>	Returns the current upper bound of a variable	p. 118
<code>getval</code>	Returns the instantiation value of a variable	p. 121
<code>is_equal</code>	Tests if two variable domains are equal	p. 129
<code>is_fixed</code>	Tests if the variable passed in argument is instantiated	p. 122
<code>is_same</code>	Tests if two decision variables represent the same variable	p. 130
<code>setdomain</code>	Sets the domain of a variable	p. 132
<code>setlb</code>	Sets the lower bound of a variable	p. 133
<code>setprecision</code>	Sets the precision relativity and value of a continuous variable	p. 136
<code>settarget</code>	Sets the target value of a variable	p. 131
<code>setub</code>	Sets the upper bound of a variable	p. 134
<code>setval</code>	Instantiate the value of a variable	p. 135

getlb

Purpose

This function returns the current lower bound of the domain of the variable passed in the argument.

Synopsis

```
function getlb(x:cpvar) : integer  
function getlb(x:cpfloatvar) : real
```

Argument

x the decision variable

Return value

The current lower bound of variable x

Example

The following example shows how to get the lower bound of a cpvar x

```
val:=getlb(x)
```

Related topics

[default_lb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#)
[getprev](#) [contains](#)

getub

Purpose

This function returns the current upper bound of the domain of the variable passed in the argument.

Synopsis

```
function getub(x:cpvar) : integer  
function getub(x:cpfloatvar) : real
```

Argument

x the decision variable

Return value

The current upper bound of x

Example

The following example shows how to get the upper bound of a cpvar x

```
val:=getub(x)
```

Related topics

[default_ub](#) [getlb](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#)
[getprev](#) [contains](#)

getmiddle

Purpose

This function returns the value nearest to the middle of the domain of the variable passed in the argument. It can be useful for specific search strategies.

Synopsis

```
function getmiddle(x:cpvar) : integer
function getmiddle(x:cpfloatvar) : real
```

Argument

x the decision variable

Return value

The middle value of a variable

Example

The following example shows how to get the middle value of a cpvar x'

```
val:=getmiddle(x)
```

Related topics

[getlb](#) [getub](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#) [contains](#)

getsize

Purpose

This function returns the cardinality (number of distinct values) of the domain of the variable passed in argument in the case of cpvar and the size of the domain interval representation of the domain in the case of a cpfloatvar.

Synopsis

```
function getsize(x:cpvar) : integer
function getsize(x:cpfloatvar) : integer
```

Argument

x the decision variable

Return value

The cardinality / size of the interval representation of the domain of x

Example

The following example shows how to get the size of the domain of a cpvar x

```
sz:=getsize(x)
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#) [contains](#)

getval

Purpose

This function returns the instantiation value of a variable passed in argument. Note that the variable must be instantiated before calling this function, else it will return its lower bound.

Synopsis

```
function getval(x:cpvar) : integer
```

Argument

x the decision variable

Return value

The instantiation value of x (or by default its lower bound)

Example

The following example shows how to get the instantiation value of a cpvar X

```
val:=getval(X)
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#) [contains](#)

is_fixed

Purpose

Returns true if the variable passed in argument has a domain reduced to a singleton.

Synopsis

```
function is_fixed(var:cpvar) :   boolean
function is_fixed(var:cpfloatvar) :   boolean
```

Argument

var the decision variable

Return value

true if var has been instantiated

Example

The following example shows how to see if a cpvar var is instantiated

```
if is_fixed(var) then
    write('value of var is ', getval(var))
end-if
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getsol](#) [getval](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#)
[contains](#)

getdegree

Purpose

Returns the degree in the constraint graph of the decision variable passed in argument (i.e., the number of constraints that involve it); this can be used for advanced search heuristics.

Synopsis

```
function getdegree(x:cpvar) : integer
function getdegree(x:cpfloatvar) : integer
```

Argument

x the decision variable

Return value

The degree of x

Example

The following example shows how to get the degree of a cpvar x

```
d:=getdegree(x)
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#) [contains](#)

gettarget

Purpose

Returns the target value (preferred value for instantiation) for the variable passed in argument

Synopsis

```
function gettarget(var:cpvar) : integer  
function gettarget(var:cpfloatvar) : real
```

Argument

var the decision variable

Return value

The target value of var

Example

The following example shows how to get the target value of a cpvar 'x'

```
tval := gettarget(x)
```

Related topics

[settarget](#) [getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [getrand](#) [getnext](#) [getprev](#)
[contains](#)

getrand

Purpose

Returns a value at random (uniform distribution) in the domain of the variable passed in argument.

Synopsis

```
function getrand(x:cpvar) : integer
function getrand(x:cpfloatvar) : real
```

Argument

x the decision variable

Return value

A random value in the domain of x

Example

The following example shows how to get a random value in the domain a cpvar x

```
r:=getrand(x)
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getnext](#) [getprev](#) [contains](#)

getnext

Purpose

Returns the value nearest to and greater than `val` in the domain of a variable passed in argument. This function is useful to enumerate the values of the domain of a variable from the lowest to the highest value; note that if 'val' is larger than the upper bound of the domain of `x`, the upper bound will be returned.

Synopsis

```
function getnext(x:cpvar,val:integer) : integer
```

Arguments

<code>x</code>	the decision variable
<code>val</code>	a value in the domain of the variable

Return value

The next value in the domain of `x`

Example

The following example shows how to enumerate in increasing order the values in the domain of a cpvar `x`

```
curVal := getlb(x)
while (curVal < getub(x)) do
    curVal := getnext(x, curVal)
    writeln("curVal= ", curVal)
end-do
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getprev](#) [contains](#)

getprev

Purpose

Returns the value nearest to and lower than 'val' in the domain of a variable passed in argument. This function is useful to enumerate the values of the domain of a variable from the highest to the lowest value; note that if 'val' is below the lower bound of x domain, the lower bound will be returned.

Synopsis

```
function getprev(x:cpvar,val:integer) : integer
```

Arguments

x	the decision variable
val	a value in the domain of the variable

Return value

The previous value in the domain of x

Example

The following example shows how to enumerate in decreasing order the values in the domain of a cpvar x

```
curVal := getub(x)
while (curVal > getlb(x)) do
    curVal := getprev(x, curVal)
    writeln("curVal= ",curVal)
end-do
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [contains](#)

contains

Purpose

Returns `true` if the value 'val' belongs to the domain of the variable passed in argument

Synopsis

```
function contains(x:cpvar,val:integer) :  boolean
function contains(x:cpfloatvar,val:real) :  boolean
```

Arguments

<code>x</code>	the decision variable
<code>val</code>	the value

Return value

`true` if `x` can be instantiated to `val`

Example

The following example shows how to test if a value belongs to the domain of a cpvar `x`

```
if contains(x,3) then
    write("x can be instantiated to three!")
end-if
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#)

is_equal

Purpose

Returns `true` if the domains of the two variables passed in argument contain exactly the same values

Synopsis

```
function is_equal(var1:cpvar,var2:cpvar) :  boolean
function is_equal(var1:cpfloatvar,var2:cpfloatvar) :  boolean
```

Arguments

`var1` the first decision variable
`var2` the second decision variable

Return value

`true` if the domain of `var1` equals the domain of `var2`, else `false`

Example

The following example shows how to test whether the domains of two `cpvar` `var1` and `var2` are equal

```
if is_equal(var1,var2) then
    write("the domains of var1 and var2 are the same!")
end-if
```

Related topics

[`is\_same`](#)

is_same

Purpose

Returns `true` if two decision variables passed in argument represent the same variable; this function is mainly used by advanced users to specify branching heuristics (see example below).

Synopsis

```
function is_same(var1:cpvar,var2:cpvar) :   boolean
function is_same(var1:cpfloatvar,var2:cpfloatvar) :   boolean
```

Arguments

`var1` the first decision variable
`var2` the second decision variable

Return value

`true` if `var1` and `var2` represent the same variable

Example

The following example shows how to test whether two `cpvar` represent the same variable

```
if is_same(var1,var2) then
    write("var1 and var2 represent the same variable!")
end-if
```

Related topics

[`is\_equal`](#)

settarget

Purpose

Sets the target value (preferred instantiation value) for the variable passed in argument.

Synopsis

```
procedure settarget(x:cpvar,value:integer)
procedure settarget(x:cpfloatvar,value:real)
```

Arguments

x	the decision variable
value	the target value

Example

The following example shows how to set the target value of a cpvar `x` to three:

```
settarget(x,3)
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#)

setdomain

Purpose

Sets the domain of the variable to the set of integers passed in argument.

Synopsis

```
procedure setdomain(x:cpvar, domain:set of integers)
procedure setdomain(x:cpvar, lowerBound:integer, upperBound:integer)
procedure setdomain(x:cpfloatvar, lowerBound:real, upperBound:real)
procedure setdomain(x:cpauxvar, lowerBound:real, upperBound:real)
```

Arguments

x	the decision variable
domain	Set of integers representing the target domain.
lowerBound	lower bound of the interval
upperBound	upper bound of the interval

Example

The following example shows how to set the domain of a cpvar `x` to the set of integers {1,3,5,7,9}

```
setdomain(x,{1,3,5,7,9})
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#)

setlb

Purpose

Sets the lower bound of the variable to the value passed in argument. This procedure can be used during the CP search.

Synopsis

```
procedure setlb(x:cpvar, value:integer)
procedure setlb(x:cpfloatvar, value:real)
```

Arguments

x	the decision variable
value	value representing the lower bound

Example

The following example shows how to set the lower bound of a cpvar `x` to the integer value 1.

```
setlb(x,1)
```

Related topics

[setval](#) [setub](#) [getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#)
[getnext](#) [getprev](#)

setub

Purpose

Sets the upper bound of the variable to the value passed in argument. This procedure can be used during the CP search.

Synopsis

```
procedure setub(x:cpvar, value:integer)
procedure setub(x:cpfloatvar, value:real)
```

Arguments

x the decision variable
value value representing the upper bound.

Example

The following example shows how to set the upper bound of a cpvar **x** to the integer value 1.

```
setub(x,1)
```

Related topics

[setval](#) [setlb](#) [getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#)
[getnext](#) [getprev](#)

setval

Purpose

Sets the value of the variable to the value passed in argument. This procedure can be used during the CP search.

Synopsis

```
procedure setval(x:cpvar, value:integer)
procedure setval(x:cpfloatvar, value:real)
```

Arguments

x	the decision variable
value	instantiation value

Example

The following example shows how to instantiate the value of a cpvar x to the integer value 1.

```
setval(x,1)
```

Related topics

[setlb](#) [setub](#) [getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [is_fixed](#) [getdegree](#) [gettarget](#) [getrand](#)
[getnext](#) [getprev](#)

setprecision

Purpose

Sets the precision relativity and value of a cpfloatvar

Synopsis

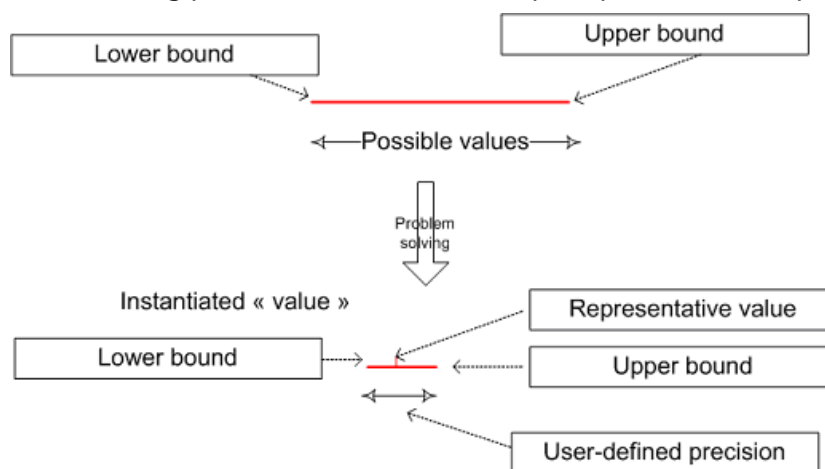
```
procedure setprecision(x:cpfloatvar, relativity:boolean, precision:real)
procedure setprecision(x:cpfloatvar, precision:real)
```

Arguments

x	the variable
relativity	relativity of precision
precision	precision value

Example

The following picture illustrates the concept of precision for a cpfloatvar



cp_show_var

Purpose

This procedure prints the current domain of the variable passed in argument

Synopsis

```
procedure cp_show_var(var:cpvar)
procedure cp_show_var(var:cpfloatvar)
```

Argument

var the decision variable to show

Related topics

[cp_show_prob](#) [cp_show_stats](#)

9.4 Problem

<code>cp_find_next_sol</code>	Finds the next solution of the problem	p. 141
<code>cp_infeas_analysis</code>	Compute a minimal conflict set for an inconsistent problem	p. 154
<code>cp_local_optimize</code>	Optimize an integer objective variable with local optimization.	p. 158
<code>cp_maximise</code>	Alias for <code>cp_maximize</code>	p. 143
<code>cp_maximize</code>	Maximizes a variable	p. 144
<code>cp_minimise</code>	Alias for <code>cp_minimize</code>	p. 145
<code>cp_minimize</code>	Minimizes a variable	p. 146
<code>cp_post</code>	Posts a constraint to the problem	p. 139
<code>cp_propagate</code>	Propagates the constraints	p. 140
<code>cp_reset_params</code>	Reset parameters to their default value.	p. 160
<code>cp_reset_search</code>	Resets the search process	p. 142
<code>cp_restore_state</code>	Restore a solver state from the stack	p. 156
<code>cp_save_state</code>	Save a marker in the stack	p. 155
<code>cp_shave</code>	Shave the variables of the problem	p. 153
<code>cp_show_best_sol</code>	Pretty printing of the best solution found.	p. 150
<code>cp_show_prob</code>	Pretty printing of the problem	p. 148
<code>cp_show_sol</code>	Pretty printing of the last solution.	p. 149
<code>getelt</code>	Retrieve the value of a reversible array element.	p. 165
<code>getname</code>	Gets the name of a variable / task / resource	p. 151
<code>getsize</code>	Returns the size of a reversible array.	p. 166
<code>getsol</code>	Returns the solution value of a variable	p. 147
<code>getval</code>	Retrieve the value of a reversible number.	p. 163
<code>path_order</code>	Return a path-order branching scheme	p. 157
<code>set_reversible_attributes</code>	Sets the attributes of a reversible number or array.	p. 161
<code>set_sol_as_target</code>	Set last solution found as target values.	p. 159
<code>setelt</code>	Set the value of an element of an array of reversibles.	p. 164
<code>setname</code>	Sets the name of a variable / task / resource	p. 152
<code>setval</code>	Set the value of a reversible number.	p. 162

cp_post

Purpose

This function posts the constraint `ctr` to the problem. Note that posting a constraint does not imply that the involved variables will be instantiated.

Synopsis

```
function cp_post(ctr:cpctr) : boolean
```

Argument

`ctr` the constraint to post

Return value

Returns `true` if the constraint is compatible with already posted constraints, else `false`.

Example

The following example shows how to post the constraint $x = y + 1$ where x and y are two decision variables of the problem

```
cp_post(x = y + 1)
```

Related topics

[cp_propagate KALIS_AUTO_PROPAGATE cp_post](#)

cp_propagate

Purpose

This function reaches a fixed point during constraint propagation. The constraint deduction rules are applied successively until no more deductions are possible. Note that propagating a constraint does not imply that the involved variables will be instantiated.

Synopsis

```
function cp_propagate : boolean
```

Return value

Returns `false` if the propagation reaches a contradiction (infeasible problem), else returns `true`

Related topics

[cp_post](#) [auto_propagate](#)

cp_find_next_sol

Purpose

Finds the next solution of the problem

Synopsis

```
function cp_find_next_sol : boolean
```

Return value

Returns `true` if an additional solution can be found, else `false`

Example

The following code shows how to scan all the solutions of a predefined problem containing `b`, `x`, `y` and `z` as decision variables:

```
while (cp_find_next_sol)
    writeln("b:", b, " x:", x, " y:", y, " z:", z)
```

Related topics

[cp_reset_search](#) [cp_minimize](#) [cp_maximize](#) [cp_set_branching](#)

cp_reset_search

Purpose

This procedure resets the search process: it finalizes the tree search and returns to the first node of the tree. While it is not called, the branch and bound stays at a node in the tree search and the problem cannot be modified. It needs to be called only if tree search was not finished.

Synopsis

```
procedure cp_reset_search
```

cp_maximise

Purpose

Alias for cp_maximize. See [cp_maximize](#) for description.

Synopsis

```
function cp_maximise(obj:cpvar) :   boolean
function cp_maximise(obj:cpfloatvar) :  boolean
```

Related topics

[cp_maximize](#)

cp_maximize

Purpose

This function starts the search for an optimal solution of the problem that maximizes a specific variable. Upon termination of the search (when no limitation has been set on the search process), the last solution found (if any exists) is proven optimal.

Synopsis

```
function cp_maximize(obj:cpvar) :  boolean
function cp_maximize :  boolean
```

Argument

obj the decision variable to maximize

Return value

true if a solution has been found, false if the problem is inconsistent (no solution exists)

Example

The following code shows how to use this function to maximize the Benefit objective

```
optimizeProcess := cp_maximize(Benefit)
```

Related topics

[cp_minimize](#) [cp_find_next_sol](#)

cp_minimise

Purpose

Alias for `cp_minimize`. See `cp_maximize` for description.

Synopsis

```
function cp_minimise(obj:cpvar) :   boolean
function cp_minimise(obj:cpfloatvar) :  boolean
```

Related topics

`cp_minimize`

cp_minimize

Purpose

This function starts the search for an optimal solution of the problem that minimizes a specific variable. Upon termination of the function (when no limitation has been set on the search process), the last solution found (if any exists) is proven optimal.

Synopsis

```
function cp_minimize(obj:cpvar) :   boolean
function cp_minimize(obj:cpfloatvar) :   boolean
```

Argument

obj the decision variable to minimize

Return value

true if a solution has been found, false if the problem is inconsistent (no solution exists)

Example

The following code shows how to use this function to minimize the objective Cost:

```
optimizeProcess := cp_minimize(Cost)
```

Related topics

[cp_maximize](#) [getsol](#) [cp_find_next_sol](#)

getsol

Purpose

This function returns the solution value of a variable or linear expression passed in argument. Note that the variable must be instantiated before calling this function, else it will return its lower bound.

Synopsis

```
function getsol(x:cpvar) : integer
function getsol(x:cpfloatvar) : real
function getsol(l:cplinear) : integer
```

Arguments

x	a decision variable
l	a linear expression

Return value

The instantiation value of x (or by default its lower bound), respectively the evaluation of the linear expression

Example

The following example shows how to get the solution value of a cpvar x

```
val:= getsol(x)
```

Related topics

[getlb](#) [getub](#) [is_fixed](#) [getsize](#)

cp_show_prob

Purpose

This procedure prints an overview of the current state of the problem (variables and constraints)

Synopsis

```
procedure cp_show_prob
```

Related topics

[cp_show_var](#) [cp_show_stats](#)

cp_show_sol

Purpose

This procedure prints an overview of the last solution found.

Synopsis

```
procedure cp_show_sol
```

Related topics

[cp_show_var](#) [cp_show_stats](#)

cp_show_best_sol

Purpose

This procedure prints an overview of the best solution found if the problem has an objective, or of the last solution found otherwise.

Synopsis

```
procedure cp_show_best_sol
```

Related topics

[cp_show_sol](#) [cp_show_var](#) [cp_show_stats](#)

getname

Purpose

Gets the name of a variable / task / resource passed in argument. The name of the modeling object is used by problem and variable printing routines

Synopsis

```
procedure getname(var:cpvar)
procedure getname(fvar:cpfloatvar)
procedure getname(task:cptask)
procedure getname(resource:cpresource)
```

Arguments

var	a finite domain variable
fvar	the floating decision variable
resource	a resource
task	a task

Example

The following example shows how to print out the name of a cpvar x

```
writeln(getname(x))
```

Related topics

[setname](#) [cp_show_prob](#) [cp_show_var](#) [cp_show_schedule](#)

setname

Purpose

Sets the name of a variable / task / resource passed in argument, the name of the variable / task / resource is used by problem and variable printing routines

Synopsis

```
procedure setname(var:cpvar,name:string)
procedure setname(fvar:cpfloatvar,name:string)
procedure setname(task:cptask,name:string)
procedure setname(resource:cpresource,name:string)
```

Arguments

var	a finite domain
fvar	a continuous variable
task	a task
resource	a resource
name	the name to set

Example

The following example shows how to set the name of a cpvar x to 'AMOUNT'

```
setname(x,"AMOUNT")
```

Related topics

[getname](#) [cp_show_prob](#) [cp_show_var](#) [cp_show_schedule](#)

cp_shave

Purpose

Shaving consists in tentatively setting variables to a value in their domain. If the assignment fails (considering the complete constraint set), then this value can be removed from the domain, possibly eliminating many inconsistent values before starting search.

Synopsis

```
function cp_shave: boolean
```

Return value

false if problem is proven to be inconsistent, true otherwise

Example

The following example shows how to apply shaving to the variables of a problem:

```
res := cp_shave
```

Related topics

[cp_propagate](#)

cp_infeas_analysis

Purpose

This method computes a minimal conflict set for an inconsistent problem. A minimal conflict set is a minimal size set of constraints that causes a contradiction (infeasibility).

Synopsis

```
procedure cp_infeas_analysis
```

Example

The following example shows how to compute a minimal conflict set:

```
...      ! definition of constraints
...
cp_save_state
if (not cp_propagate) then
  cp_restore_state
  writeln("Problem is infeasible")
  cp_infeas_analysis
  exit(0)
end-if
cp_restore_state
...      ! problem solving
```

Related topics

[cp_save_state](#) [cp_restore_state](#)

cp_save_state

Purpose

Save a marker in the CP solver stack to enable restoration of the domains of decision variables at a later stage, e.g., for performing infeasibility analysis.

Synopsis

```
procedure cp_save_state
```

Example

The following example shows how to save the state of a constraint system:

```
...      ! definition of constraints
...
cp_save_state
if (not cp_propagate) then
  cp_restore_state
  writeln("Problem is infeasible")
  cp_infeas_analysis
  exit(0)
end-if
cp_restore_state
...      ! problem solving
```

Related topics

[cp_restore_state](#)

cp_restore_state

Purpose

Restore the state of the CP solver at a previously saved state marker. Note: this function only restores the domains of decision variables, constraints that have been posted after saving the state marker are not removed.

Synopsis

```
procedure cp_restore_state
```

Example

The following example shows how to restore the state of a constraint system:

```
...      ! definition of constraints
...
cp_save_state
if (not cp_propagate) then
  cp_restore_state
  writeln("Problem is infeasible")
  cp_infeas_analysis
  exit(0)
end-if
cp_restore_state
...      ! problem solving
```

Related topics

[cp_save_state](#)

path_order

Purpose

The path-heuristic is specifically designed to work in conjunction with the cycle constraint. It selects the first unassigned 'succ' variable in the order of the current subpath implied by the current values of the successors variables. When the subpath is empty (no 'succ' variables are currently instantiated) the path order-heuristic selects a node at random.

Synopsis

```
function path_order(succ:array of cpvar, nodeSelection:string) : cpbranching
```

Arguments

succ	The list of successors variables
nodeSelection	The value selection heuristic

Related topics

[cycle](#)

cp_local_optimize

Purpose

This function starts the search for a near-optimal solution of the problem that optimizes a specific objective variable.

Synopsis

```
function cp_local_optimize(obj:cpvar, minimize: integer) : boolean  
function cp_local_optimize(obj:cpfloatvar, minimize: integer) : boolean
```

Arguments

obj	the objective variable
minimize	0 for minimization or 1 for maximization

Related topics

[cp_minimize](#) [cp_maximize](#)

set_sol_as_target

Purpose

Set the target values of all the variables to their value in the last solution found.

Synopsis

```
procedure set_sol_as_target
```

Related topics

[gettarget settarget KALIS_NEAREST_VALUE](#)

cp_reset_params

Purpose

Reset parameters to their default value.

Synopsis

```
procedure cp_reset_params(parameterSet:integer)
```

Argument

parameterSet	the set of parameters to reset (KALIS_RESET_PARAMS_ALL , KALIS_RESET_VAR_BOUNDS , KALIS_RESET_VAR_PRECISION , KALIS_RESET_OPT_PARAMS , KALIS_RESET_SEARCH_PARAMS)
--------------	---

Example

The following example shows how to use the `cp_reset_params` function to reset all parameters of Xpress Kalis:

```
cp_reset_params(KALIS_RESET_PARAMS_ALL)
```

Related topics

[setparam](#)

set_reversible_attributes

Purpose

Sets the attributes of a reversible number or array.

Synopsis

```
procedure set_reversible_attributes(revarray:cpreversiblearray, first:integer,
                                   last:integer, initValue:integer)
procedure set_reversible_attributes(rev:cpreversible, initValue:real)
```

Arguments

revarray	a reversible array
first	the smallest index number
last	the highest index number
rev	a reversible number
initValue	the initialization value

Example

The following example shows how to set the attributes of a reversible array :

```
declarations
  reva: cpreversiblearray
end-declarations

! Create a reversible array of 10 elements initialized with value 0
set_reversible_attributes(reva, 1, 10, 0)

! Return the number of elements in the reversible array
writeln("Array contains ", getsize(reva), " elements")

! Get the 5'th element of the array
writeln("The 5'th element of reva is ", getelt(reva,5))
```

Related topics

[setval](#) [getval](#) [getsize](#) [getelt](#) [setelt](#)

setval

Purpose

Sets the value of a reversible number. If this procedure is used during the CP search, the reversible will be restored to its previous value on backtracking beyond the state where this value has been set.

Synopsis

```
procedure setval(k:cpreversible, value:real)
```

Arguments

k a reversible number,
value instantiation value.

Example

The following example shows how to set the current value of a reversible `rev` to the integer value 10.

```
setval(rev, 10)
```

Related topics

[set_reversible_attributes](#) [getval](#)

getval

Purpose

Retrieves the value of a reversible number at the current state of the constraint system.

Synopsis

```
function getval(k:cpreversible): real)
```

Argument

k a reversible number.

Return value

The current value of the reversible.

Example

The following example saves the state of the constrain system, changes the value of the reversible `rev`, then restores the previous state of th constraint system and displays the restored value of the reversible number.

```
set_reversible_attributes(rev, 20.54)

! Save current state of constraint system
cp_save_state

setval(rev, 5)
writeln("Value of the reversible is ", getval(rev))

! Restore the saved system state
cp_restore_state
writeln("After state restoration reversible is ", rev)
```

Related topics

[set_reversible_attributes](#) [setval](#)

setelt

Purpose

Sets the value of an element of an array of reversibles. If this procedure is used during the CP search, the reversible will be restored to its previous value on backtracking beyond the state where this value has been set.

Synopsis

```
procedure setelt(ra:cpreversiblearray, ind: integer, value:integer)
```

Arguments

ra	an array of reversibles,
ind	the index of an array element,
value	new value for the reversible.

Example

The following example saves the state of the constraint system, changes the value of an element the reversible array `reva`, then restores the previous state of the constraint system and displays the restored value of the reversible number.

```
set_reversible_attributes(reva, 1, 10, 0)

! Save current state of constraint system
cp_save_state

setelt(reva, 4, -10)
writeln("The 4'th element of 'reva' is ", setelt(reva, 4))

! Revert to the saved system state
cp_restore_state
writeln("After state restoration reversible array is: ", reva)
```

Related topics

[set_reversible_attributes](#) [getelt](#) [getsize](#)

getelt

Purpose

Retrieves the value of the specified element of an array of reversible numbers at the current state of the constraint system.

Synopsis

```
function getelt(ra:cpreversiblearray, ind:integer): integer)
```

Arguments

ra	an array of reversibles,
ind	the index of an array element.

Return value

The current value of the specified entry of the array.

Example

See [setelt](#).

Related topics

[set_reversible_attributes](#) [setelt](#) [getsize](#)

getsize

Purpose

This function returns the size (= number of elements) of a reversible array.

Synopsis

```
function getsize(ra:cpreversiblearray): integer
```

Argument

ra the array of reversibles

Return value

The size of the array of reversibles.

Example

The following example shows how to get the size of the array reva.

```
set_reversible_attributes(reva, 1, 10, 0)
writeln("Size of array 'reva': ", reva.size)
```

Related topics

[set_reversible_attributes](#) [getelt](#) [setelt](#)

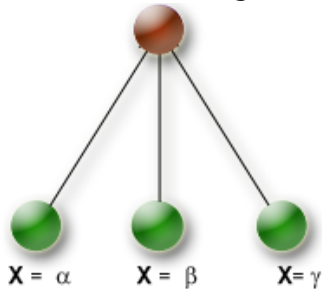
9.5 Search

<code>assign_and_forbid</code>	assign_and_forbid branching scheme	p. 172
<code>assign_var</code>	assign_var branching scheme	p. 168
<code>bs_group</code>	Create a group of branching schemes	p. 193
<code>cp_set_branching</code>	Sets the strategy to use during the search for a solution	p. 189
<code>cp_show_stats</code>	Shows some statistics about the search	p. 192
<code>gettag</code>	Gets the tag associated with a branching scheme group	p. 195
<code>group_serializer</code>	Creates a branching scheme Group Serializer	p. 194
<code>probe_assign_var</code>	probe_assign_var branching scheme	p. 178
<code>probe_settle_disjunction</code>	probe_settle_disjunction branching scheme	p. 179
<code>settle_disjunction</code>	settle_disjunction branching scheme	p. 176
<code>split_domain</code>	split_domain branching scheme	p. 181
<code>task_serialize</code>	task_serialize branching scheme	p. 185

assign_var

Purpose

Creates an assign_var (Constraint programming style) branching scheme; in this scheme, a discrete value is assigned to the branching variable at each branch.



Synopsis

```

function assign_var(varsel:string,valsels:string,variables:set of cpvar) : cpbranching
function assign_var(varsel:string,valsels:string,variables:array(range) of cpvar) :
    cpbranching
function assign_var(varsel:string,valsels:string,variables:cpvarlist) : cpbranching
function assign_var(varsel:string,valsels:string) : cpbranching
  
```

Arguments

varsel	name of the variable selector
valsels	name of the value selector
variables	list of variables to branch on

Return value

assign_var branching scheme with default value and variable selector

Example

The following example shows how to use an assign_var branching scheme during the search process

```

model "User branching"
  uses "kalis"

  parameters
    ALG=1
  end-parameters

  forward function varchoice(Vars: cpvarlist): integer
  forward function varchoice2(Vars: cpvarlist): integer
  forward function valchoice(x: cpvar): integer
  forward function valchoice2(x: cpvar): integer

  setparam("KALIS_DEFAULT_LB", 0);
  setparam("KALIS_DEFAULT_UB", 20)

  declarations
    R = 1..10
    y: array(R) of cpvar
    C: array(R) of integer
    Strategy: array(range) of cpbranching
  end-declarations
  
```

```

C:: [4, 7, 2, 6, 9, 0,-1, 3, 8,-2]

all_different(y)
forall(i in R | isodd(i)) y(i) >= y(i+1) + 1
y(4) + y(1) = 13; y(8) <= 15; y(7) <> 5

! Definition of user branching strategies:
Strategy(1):= assign_and_forbid("varchoice2", "valchoice", y)
Strategy(2):= assign_var("varchoice", "valchoice", y)
Strategy(3):= split_domain("varchoice", "valchoice2", y, true, 2)
Strategy(4):= split_domain("varchoice2", "valchoice", y, false, 5)

! Select a branching strategy
cp_set_branching(Strategy(ALG))

if cp_find_next_sol then
  forall(i in R) write(getsol(y(i)), " ")
  writeln
end-if

!-----
! **** Variable choice ****
! **** Choose variable with largest degree + smallest domain
function varchoice(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
  end-declarations

  ! Get the number of elements of "Vars"
  listsize:= getsize(Vars)

  ! Set on uninstantiated variables
  forall(i in 1..listsize)
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if Vset={} then
    returned:= 0
  else
    ! Get the variables with max. degree
    dmax:= max(i in Vset) getdegree(getvar(Vars,i))
    forall(i in Vset)
      if getdegree(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
    dsize:= MAX_INT

    ! Choose var. with smallest domain among those indexed by 'Iset'
    forall(i in Iset)
      if getsize(getvar(Vars,i)) < dsize then
        returned:= i
        dsize:= getsize(getvar(Vars,i))
      end-if
    end-if
    writeln(returned)
  end-function

```

```

! **** Choose variable y(i) with smallest value of C(i)
function varchoice2(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
    VarInd: array(Iset) of integer
  end-declarations

! Set on uninstantiated variables
  listsize:= getsize(Vars)
  forall(i in 1..listsize)
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if getsize(Vset)=0 then
    returned:= 0
  else
! Establish a correspondence of indices between 'Vars' and 'y'
    forall(i in R)
      forall(j in Vset)
        if is_same(getvar(Vars,j), y(i)) then
          VarInd(i):= j
          Vset -= {j}
          break 1
        end-if

! Choose the variable
    imin:= min(i in Iset) i; cmin:= C(imin)
    forall(i in Iset)
      if C(i) < cmin then
        imin:= i; cmin:= C(i)
      end-if
    returned:= VarInd(imin)
  end-if
  writeln(imin, " ", returned)
end-function

!-----
! *** Value choice ****
! **** Choose the next value one third larger than lower bound
! (Strategy may be used with any branching scheme since it
! makes sure that the chosen value lies in the domain)
function valchoice(x: cpvar): integer
! returned:= getlb(x)
  returned:= getnext(x, getlb(x) + round((getub(x)-getlb(x))/3))
  writeln("Value: ", returned, " ", contains(x,returned),
    " x: ", x)
end-function

! **** Split the domain into lower third and upper two thirds
! (Strategy to be used only with 'split_domain' branching since
! the chosen value may not be in the domain)
function valchoice2(x: cpvar): integer
  returned:= getlb(x) + round((getub(x)-getlb(x))/3)
  writeln("Value: ", returned, " x: ", x)
end-function
end-model

```

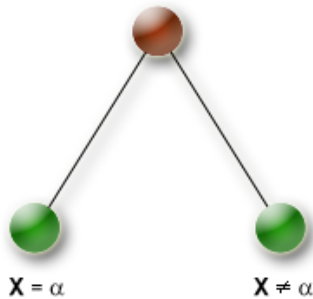
Related topics

[assign_and_forbid](#) [settle_disjunction](#) [probe_assign_var](#) [split_domain](#)

assign_and_forbid

Purpose

Creates an assign_and_forbid (MIP style) branching scheme; that will assign a value to the branching variable on one branch and forbid that value on the other branch.



Synopsis

```
function assign_and_forbid(varsel:string,valsels:string,variables:set of cpvar) :
    cpbranching
function assign_and_forbid(varsel:string,valsels:string,variables:array(range) of
    cpvar) : cpbranching
function assign_and_forbid(varsel:string,valsels:string,variables:cpvarlist) :
    cpbranching
function assign_and_forbid(varsel:string,valsels:string) : cpbranching
```

Arguments

varsel	name of the variable selector
valsels	name of the value selector
variables	list of variables to branch on

Return value

assign_and_forbid branching scheme with value selector 'valsels' and variable selector 'varsels' working on all specified variables

Example

The following example shows how to use an assign_var branching scheme during the search process

```
model "User branching"
uses "kalis"

parameters
    ALG=1
end-parameters

forward function varchoice(Vars: cpvarlist): integer
forward function varchoice2(Vars: cpvarlist): integer
forward function valchoice(x: cpvar): integer
forward function valchoice2(x: cpvar): integer

setparam("KALIS_DEFAULT_LB", 0);
setparam("KALIS_DEFAULT_UB", 20)

declarations
    R = 1..10
```

```

    y: array(R) of cpvar
    C: array(R) of integer
    Strategy: array(range) of cpbranching
end-declarations

C:: [4, 7, 2, 6, 9, 0,-1, 3, 8,-2]

all_different(y)
forall(i in R | isodd(i)) y(i) >= y(i+1) + 1
y(4) + y(1) = 13; y(8) <= 15; y(7) <> 5

! Definition of user branching strategies:
Strategy(1):= assign_and_forbid("varchoice2", "valchoice", y)
Strategy(2):= assign_var("varchoice", "valchoice", y)
Strategy(3):= split_domain("varchoice", "valchoice2", y, true, 2)
Strategy(4):= split_domain("varchoice2", "valchoice", y, false, 5)

! Select a branching strategy
cp_set_branching(Strategy(ALG))

if cp_find_next_sol then
    forall(i in R) write(getsol(y(i)), " ")
    writeln
end-if

!-----
! **** Variable choice ****
! **** Choose variable with largest degree + smallest domain
function varchoice(Vars: cpvarlist): integer
    declarations
        Vset,Iset: set of integer
    end-declarations

    ! Get the number of elements of "Vars"
    listsize:= getsize(Vars)

    ! Set on uninstantiated variables
    forall(i in 1..listsize)
        if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

    if Vset={} then
        returned:= 0
    else
        ! Get the variables with max. degree
        dmax:= max(i in Vset) getdegree(getvar(Vars,i))
        forall(i in Vset)
            if getdegree(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
        dsize:= MAX_INT

        ! Choose var. with smallest domain among those indexed by 'Iset'
        forall(i in Iset)
            if getsize(getvar(Vars,i)) < dsize then
                returned:= i
                dsize:= getsize(getvar(Vars,i))
            end-if
    end-if
end-function

```

```

end-if
writeln(returned)
end-function

! **** Choose variable y(i) with smallest value of C(i)
function varchoice2(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
    VarInd: array(Iset) of integer
  end-declarations

  ! Set on uninstantiated variables
  listsize:= getsize(Vars)
  forall(i in 1..listsize)
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if getsize(Vset)=0 then
    returned:= 0
  else
    ! Establish a correspondence of indices between 'Vars' and 'y'
    forall(i in R)
      forall(j in Vset)
        if is_same(getvar(Vars,j), y(i)) then
          VarInd(i):= j
          Vset -= {j}
          break 1
        end-if
      end-if

    ! Choose the variable
    imin:= min(i in Iset) i; cmin:= C(imin)
    forall(i in Iset)
      if C(i) < cmin then
        imin:= i; cmin:= C(i)
      end-if
    returned:= VarInd(imin)
  end-if
  writeln(imin, " ", returned)
end-function

!-----
! *** Value choice ****
! **** Choose the next value one third larger than lower bound
! (Strategy may be used with any branching scheme since it
! makes sure that the chosen value lies in the domain)
function valchoice(x: cpvar): integer
! returned:= getlb(x)
  returned:= getnext(x, getlb(x) + round((getub(x)-getlb(x))/3))
  writeln("Value: ", returned, " ", contains(x,returned),
    " x: ", x)
end-function

! **** Split the domain into lower third and upper two thirds
! (Strategy to be used only with 'split_domain' branching since
! the chosen value may not be in the domain)

```

```
function valchoice2(x: cvar): integer
  returned:= getlb(x) + round((getub(x)-getlb(x))/3)
  writeln("Value: ", returned, " x: ", x)
end-function
end-model
```

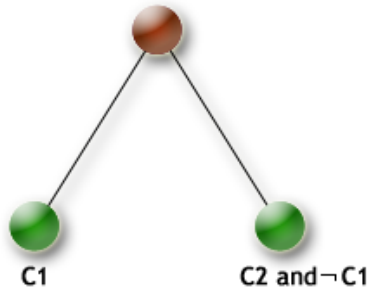
Related topics

[assign_var](#) [settle_disjunction](#) [probe_assign_var](#) [split_domain](#)

settle_disjunction

Purpose

Creates a `settle_disjunction` branching scheme that resolves the status of all the disjunctions passed in argument. The branching consists in choosing one branch of the disjunction and posting the constraint stated by this branch. The branches are tested from left to right



Synopsis

```

function settle_disjunction(constraints:set of cpctr) : cpbranching
function settle_disjunction(constraints:array(range) of cpctr) : cpbranching
function settle_disjunction(constraints:set of cpctr) : cpbranching
function settle_disjunction(constraints:array(range) of cpctr) : cpbranching
function settle_disjunction : cpbranching
function settle_disjunction : cpbranching
  
```

Argument

`constraints` the disjunctions

Return value

The resulting `settle_disjunction` branching scheme

Example

The following example shows how to use the `settle_disjunction` branching scheme to solve a small disjunctive scheduling problem: The problem is to find a schedule for a set of tasks on one machine. The machine can process only one task at the time and the goal is to minimize the total weighted tardiness of the schedule.

```

model "Disjunctive scheduling with settle_disjunction"
uses "kalis"

declarations
  NBTASKS = 5
  TASKS = 1..NBTASKS
  DUR: array(TASKS) of integer
  DUE: array(TASKS) of integer
  WEIGHT: array(TASKS) of integer
  start: array(TASKS) of cpvar
  tmp: array(TASKS) of cpvar
  tardiness: array(TASKS) of cpvar
  twt: cpvar
  zeroVar: cpvar
  Strategy: array(range) of cpbranching
  Disj: set of cpctr
end-declarations

DUR :: [21,53,95,55,34]
  
```

```
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

forall (t in TASKS) do
  start(t) >= 0
  setname(start(t), "Start("+t+")")
  tmp(t) = start(t) + DUR(t) - DUE(t)
  setname(tardiness(t), "Tard("+t+")")
  tardiness(t) = maximum({tmp(t),zeroVar})
end-do

twt = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
forall(t in 1..NBTASKS-1, s in t+1..NBTASKS)
  (start(t) + DUR(t) <= start(s)) or
  (start(s) + DUR(s) <= start(t))

! Define the branching strategy
Strategy(1):= settle_disjunction
Strategy(2):= split_domain(KALIS_LARGEST_MIN,KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

! Solve the problem
if not(cp_minimize(twt)) then
  writeln("Problem is inconsistent")
  exit(0)
end-if

forall (t in TASKS)
  writeln("[", getsol(start(t)), "==>",
    getsol(start(t)) + DUR(t), "]:\t ",
    getsol(tardiness(t)), "  (", getsol(tmp(t)), ")")
  writeln("Total weighted tardiness: ", getsol(twt))

end-model
```

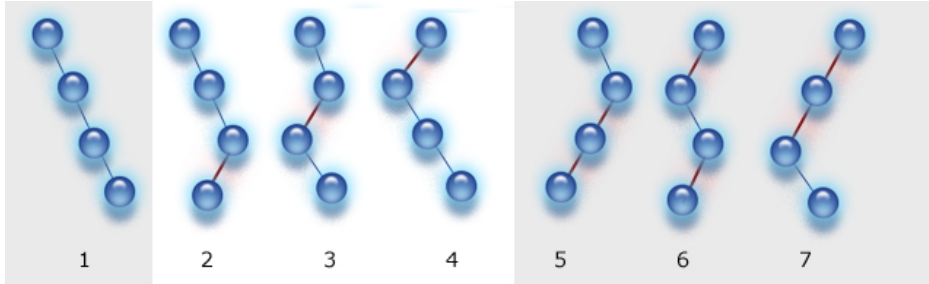
Related topics

[assign_var](#) [assign_and_forbid](#) [probe_assign_var](#) [split_domain](#)

probe_assign_var

Purpose

Creates a `probe_assign_var` branching scheme, in which probing and assignment are applied simultaneously



Synopsis

```
function probe_assign_var(varsel:string, valsel:string, variables:set of cpvar,
    probelevel:integer) : cpbranching
function probe_assign_var(varsel:string, valsel:string, variables:array(range) of
    cpvar, probelevel:integer) : cpbranching
function probe_assign_var(varsel:string, valsel:string, variables:cpvarlist,
    probelevel:integer) : cpbranching
function probe_assign_var(varsel:string, valsel:string, probelevel:integer) :
    cpbranching
function probe_assign_var(varsel:string, valsel:string) : cpbranching
```

Arguments

<code>varsel</code>	the variable selector name (pre-defined constant or user-defined function name)
<code>valsel</code>	the value selector (pre-defined constant or user-defined function name)
<code>variables</code>	list of variables to branch on
<code>probelevel</code>	maximal probing level

Return value

The resulting `probe_assign_var` branching scheme

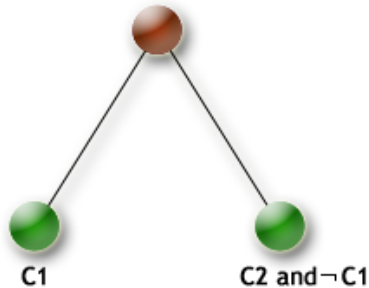
Related topics

[assign_var](#) [assign_and_forbid](#) [settle_disjunction](#) [split_domain](#)

probe_settle_disjunction

Purpose

Creates a `probe_settle_disjunction` branching scheme that resolves the status of all the disjunctions passed in argument. The branching consists in choosing one branch of the disjunction and posting the constraint stated by this branch. The branches are tested from left to right



Synopsis

```

function probe_settle_disjunction(disjunctions:set of cpctr, probeLevel:integer) :
    cpbranching
function probe_settle_disjunction(disj_selector:string, disjunctions:set of cpctr,
    probeLevel:integer) : cpbranching
function probe_settle_disjunction(disj_selector:string, disjunctions:array(range) of
    cpctr, probeLevel:integer) : cpbranching
function probe_settle_disjunction(disjunctions:array(range) of cpctr,
    probeLevel:integer) : cpbranching
function probe_settle_disjunction(disj_selector:string, probeLevel:integer) :
    cpbranching
function probe_settle_disjunction(probeLevel:integer) : cpbranching
  
```

Arguments

<code>disj_selector</code>	the disjunction selector name (pre-defined constant or user-defined function name)
<code>disjunctions</code>	the set or array of disjunctions
<code>probeLevel</code>	maximal probing level

Return value

The resulting `probe_settle_disjunction` branching scheme

Example

The following example shows how to use the `probe_settle_disjunction` branching scheme to solve a small disjunctive scheduling problem: The problem consists of finding a schedule for some tasks on one machine. The machine can process only one task at the time and the goal is to minimize the total weighted tardiness of the schedule. Note that the result may be (and will be in this case) suboptimal as the search tree is not fully explored.

```

model "Disjunctive scheduling with probe_settle_disjunction"
uses "kalis"

declarations
    NBTASKS = 5
    TASKS = 1..NBTASKS
    DUR: array(TASKS) of integer
    DUE: array(TASKS) of integer
    WEIGHT: array(TASKS) of integer
    ! Set of tasks
    ! Task durations
    ! Due dates
    ! Weights of tasks
  
```

```

start: array(TASKS) of cpvar      ! Start times
tmp: array(TASKS) of cpvar       ! Aux. variable
tardiness: array(TASKS) of cpvar ! Tardiness
tw: cpvar                       ! Objective variable
zeroVar: cpvar                   ! 0-valued variable
Strategy: array(range) of cpbranching ! Branching strategy
Disj: set of cpctr               ! Disjunctions
end-declarations

DUR :: [21,53,95,55,34]
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

forall(t in TASKS) do
  start(t) >= 0
  start(t).name:= "Start("+t+")"
  tmp(t) = start(t) + DUR(t) - DUE(t)
  tardiness(t).name:= "Tard("+t+")"
  tardiness(t) = maximum({tmp(t),zeroVar})
end-do

tw = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
forall(t in 1..NBTASKS-1, s in t+1..NBTASKS)
  (start(t) + DUR(t) <= start(s)) or
  (start(s) + DUR(s) <= start(t))

! Define the branching strategy
Strategy(1):= probe_settle_disjunction(1)
Strategy(2):= split_domain(KALIS_LARGEST_MIN,KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

! Solve the problem
if not(cp_minimize(twt)) then
  writeln("problem is inconsistent")
  exit(0)
end-if

forall (t in TASKS)
  writeln("[", start(t).sol, "=>",
    start(t).sol + DUR(t), "]:\t ",
    tardiness(t).sol, "  (", tmp(t).sol, ")")
  writeln("Total weighted tardiness: ", tw.sol)

end-model

```

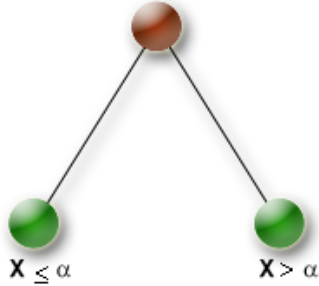
Related topics

[settle_disjunction](#) [probe_assign_var](#) [split_domain](#)

split_domain

Purpose

Creates a split_domain branching scheme. This will split the domain of every branching variable X in two parts, one branch with the case $X \leq \alpha$ and the other branch $X > \alpha$ where α is a value in the domain of X .



Synopsis

```

function split_domain(varsel:string, valsel:string, variables:set of cpvar,
    leqfirst:boolean, stopsplit: integer) : cpbranching
function split_domain(varsel:string, valsel:string, variables:array(range) of cpvar,
    leqfirst:boolean, stopsplit: integer) : cpbranching
function split_domain(varsel:string, valsel:string, variables:cpvarlist,
    leqfirst:boolean, stopsplit: integer) : cpbranching
function split_domain(varsel:string, valsel:string, leqfirst:boolean, stopsplit:
    integer) : cpbranching
function split_domain(varsel:string, valsel:string) : cpbranching
function split_domain(varsel:string, valsel:string, variables:array(range) of cpvar,
    leqfirst:boolean, stopsplit: integer, probeLevel:integer) : cpbranching
function split_domain(varsel:string, valsel:string, variables:set of cpfloatvar,
    leqfirst:boolean, stopsplit: integer) : cpbranching
function split_domain(varsel:string, valsel:string, variables:array(range) of
    cpfloatvar, leqfirst:boolean, stopsplit: integer) : cpbranching
  
```

Arguments

varsel	the variable selector name (pre-defined constant or user-defined function name)
valsel	the value selector (pre-defined constant or user-defined function name)
variables	list of variables to branch on
leqfirst	Explore the case $\leq$ first if true
stopsplit	stop branching if the size of the domain of the variable is less than stopsplit

Return value

split_domain branching scheme

Example

The following example shows how to use a split_domain branching scheme during the search

```

model "User branching"
uses "kalis"

parameters
  ALG=1
end-parameters

forward function varchoice(Vars: cpvarlist): integer
forward function varchoice2(Vars: cpvarlist): integer
  
```

```

forward function valchoice(x: cpvar): integer
forward function valchoice2(x: cpvar): integer

setparam("KALIS_DEFAULT_LB", 0);
setparam("KALIS_DEFAULT_UB", 20)

declarations
  R = 1..10
  y: array(R) of cpvar
  C: array(R) of integer
  Strategy: array(range) of cpbranching
end-declarations

C:: [4, 7, 2, 6, 9, 0,-1, 3, 8,-2]

all_different(y)
forall(i in R | isodd(i)) y(i) >= y(i+1) + 1
y(4) + y(1) = 13; y(8) <= 15; y(7) <> 5

! Definition of user branching strategies:
Strategy(1):= assign_and_forbid("varchoice2", "valchoice", y)
Strategy(2):= assign_var("varchoice", "valchoice", y)
Strategy(3):= split_domain("varchoice", "valchoice2", y, true, 2)
Strategy(4):= split_domain("varchoice2", "valchoice", y, false, 5)

! Select a branching strategy
cp_set_branching(Strategy(ALG))

if cp_find_next_sol then
  forall(i in R) write(getsol(y(i)), " ")
  writeln
end-if

!-----
! **** Variable choice ****
! **** Choose variable with largest degree + smallest domain
function varchoice(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
  end-declarations

  ! Get the number of elements of "Vars"
  listsize:= getsize(Vars)

  ! Set on uninstantiated variables
  forall(i in 1..listsize)
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if Vset={} then
    returned:= 0
  else
    ! Get the variables with max. degree
    dmax:= max(i in Vset) getdegree(getvar(Vars,i))
    forall(i in Vset)
      if getdegree(getvar(Vars,i)) = dmax then Iset+= {i}; end-if

```

```

    dsize:= MAX_INT

    ! Choose var. with smallest domain among those indexed by 'Iset'
    forall(i in Iset)
        if getsize(getvar(Vars,i)) < dsize then
            returned:= i
            dsize:= getsize(getvar(Vars,i))
        end-if
    end-if
    writeln(returned)
end-function

! **** Choose variable y(i) with smallest value of C(i)
function varchoice2(Vars: cpvarlist): integer
    declarations
        Vset,Iset: set of integer
        VarInd: array(Iset) of integer
    end-declarations

    ! Set on uninstantiated variables
    listsize:= getsize(Vars)
    forall(i in 1..listsize)
        if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

    if getsize(Vset)=0 then
        returned:= 0
    else
        ! Establish a correspondence of indices between 'Vars' and 'y'
        forall(i in R)
            forall(j in Vset)
                if is_same(getvar(Vars,j), y(i)) then
                    VarInd(i):= j
                    Vset -= {j}
                    break 1
                end-if
            end-if
        end-if

        ! Choose the variable
        imin:= min(i in Iset) i; cmin:= C(imin)
        forall(i in Iset)
            if C(i) < cmin then
                imin:= i; cmin:= C(i)
            end-if
        end-if
        returned:= VarInd(imin)
    end-if
    writeln(imin, " ", returned)
end-function

!-----
! *** Value choice ****
! **** Choose the next value one third larger than lower bound
! (Strategy may be used with any branching scheme since it
! makes sure that the chosen value lies in the domain)
function valchoice(x: cpvar): integer
! returned:= getlb(x)

```



```
    returned:= getnext(x, getlb(x) + round((getub(x)-getlb(x))/3))
    writeln("Value: ", returned, " ", contains(x,returned),
           " x: ", x)
end-function

! **** Split the domain into lower third and upper two thirds
! (Strategy to be used only with 'split_domain' branching since
! the chosen value may not be in the domain)
function valchoice2(x: cvar): integer
    returned:= getlb(x) + round((getub(x)-getlb(x))/3)
    writeln("Value: ", returned, " x: ", x)
end-function
end-model
```

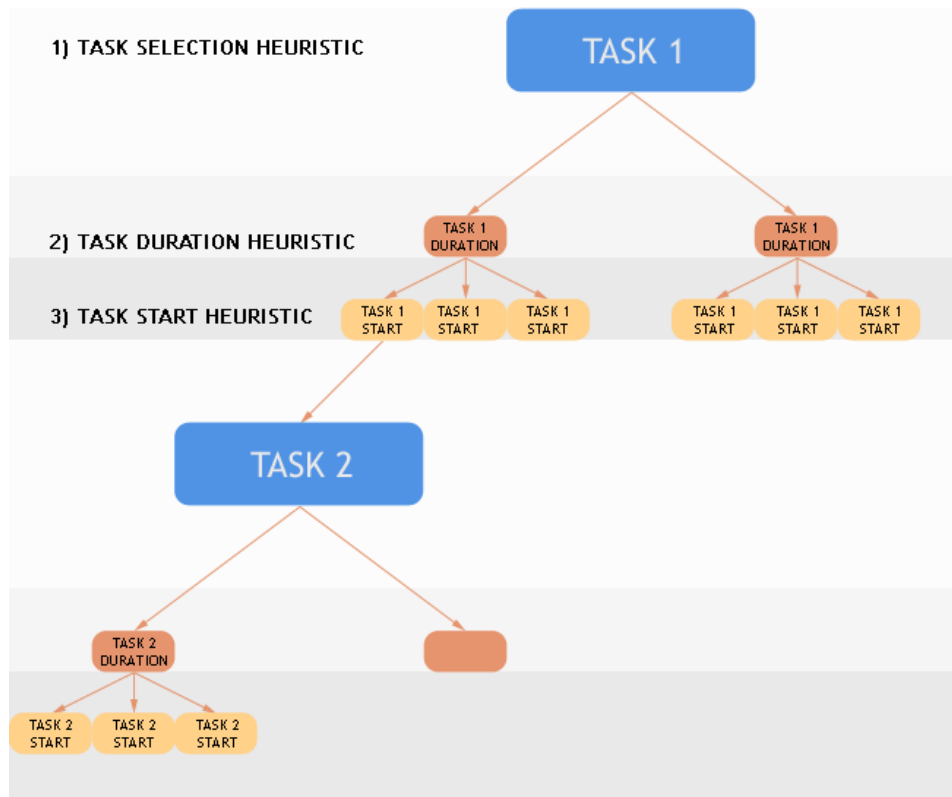
Related topics

[assign_var](#) [assign_and_forbid](#) [probe_assign_var](#) [settle_disjunction](#)

task_serialize

Purpose

Creates a Task Serializer branching scheme that serializes the tasks passed in argument. The branching consists in choosing one task to schedule. Then a branch is created for each possible duration for this task. Once the duration is determined a branch is created for each possible starting values for this task. When the start and the duration are fixed, the task is scheduled and the process is repeated until all the tasks are serialized.



Synopsis

```
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:set of cptask) : cpbranching
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:array of cptask) : cpbranching
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:array of cptask, limit:integer) : cpbranching
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:array of cptask, limit:integer) : cpbranching
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:array of cptask, limit:integer) : cpbranching
function task_serialize(taskSelector:string, durationHeuristic:string,
    startHeuristic:string, tasks:array of cptask, limit:integer) : cpbranching
```

Arguments

taskSelector	the task selector
durationHeuristic	the task duration assignment heuristic
startHeuristic	the task start time assignment heuristic
tasks	the set of tasks to be serialized.
limit	discrepancy limit: value 0 means strictly following the branching scheme, positive values indicate the permissible number of deviations from the branching scheme.

Return value

The resulting Task Serializer branching scheme

Example

The following example shows how to use the task_serialize branching scheme to solve a small cumulative scheduling problem:

```
*****!)
model "Tasks serialization example"
  uses "kalis"

  declarations
    Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
    Ceiling, Painting, MovingIn : cptask    ! Declaration of tasks
    taskset : set of cptask
    money_available : cresource              ! Resource declaration
  end-declarations

  forward function selectNextTask(tasks: cptasklist) : integer

  ! 'money_available' is a cumulative resource with max. amount of 29$
  set_resource_attributes(money_available, KALIS_DISCRETE_RESOURCE, 29)

  ! Limit resource availability to 20$ in the time interval [0,14]
  setcapacity( money_available, 0, 14, 20 )

  ! Setting the task durations and predecessor sets
  set_task_attributes(Masonry , 7 )
  set_task_attributes(Carpentry, 3, {Masonry} )
  set_task_attributes(Roofing , 1, {Carpentry} )
  set_task_attributes(Windows , 1, {Roofing} )
  set_task_attributes(Facade , 2, {Roofing} )
  set_task_attributes(Garden , 1, {Roofing} )
  set_task_attributes(Plumbing , 8, {Masonry} )
  set_task_attributes(Ceiling , 3, {Masonry} )
  set_task_attributes(Painting , 2, {Ceiling} )
  set_task_attributes(MovingIn , 1, {Windows, Facade, Garden, Painting})

  ! Setting the resource consumptions
  consumes(Masonry , 7, money_available )
  consumes(Carpentry, 3, money_available )
  consumes(Roofing , 1, money_available )
  consumes(Windows , 1, money_available )
  consumes(Facade , 2, money_available )
  consumes(Garden , 1, money_available )
  consumes(Plumbing , 8, money_available )
```

```

consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Set of tasks to schedule
taskset := {Masonry, Carpentry, Roofing, Windows, Facade, Garden,
           Plumbing, Ceiling, Painting, MovingIn}

! Set the custom branching strategy using task_serialize:
! - the task serialization process will use the function
!   "selectNextTask" to look for the next task to fix
! - it will use the "KALIS_MAX_TO_MIN" value selection heuristic
!   to set the tasks duration variable
! - and the "KALIS_MIN_TO_MAX" value selection heuristic to set
!   the start of the task
cp_set_branching(task_serialize("selectNextTask",
                                KALIS_MAX_TO_MIN, KALIS_MIN_TO_MAX, taskset))

! Find the optimal schedule (minimizing the makespan)
if (0 <> cp_schedule(getmakespan)) then
    cp_show_sol
else
    writeln("No solution found")
end-if

!-----
! **** Function to select the next task to schedule
function selectNextTask(tasks: cptasklist) : integer
    write("selectNextTask : ")
    declarations
        Vset,Iset: set of integer
    end-declarations

    ! Get the number of elements of "tasks"
    listsize:= getsize(tasks)

    ! Set of uninstantiated variables
    forall(i in 1..listsize)
        if not is_fixed(getstart(gettask(tasks,i))) or
           not is_fixed(getduration(gettask(tasks,i))) then
            Vset+= {i};
        end-if

    if Vset={} then
        returned:= 0
    else
        ! Get the variables with max. degree
        dmax:= max(i in Vset) getsize(getduration(gettask(tasks,i)))
        forall(i in Vset)
            if getsize(getduration(gettask(tasks,i))) = dmax then
                Iset+= {i}; end-if
        dsize:= MAX_INT

        ! Choose var. with smallest domain among those indexed by 'Iset'
        forall(i in Iset)

```

```
        if getsize(getstart(gettask(tasks,i))) < dsize then
            returned:= i
            dsize:= getsize(getstart(gettask(tasks,i)))
        end-if
    end-if

    if (returned <> 0) then
        writeln(gettask(tasks,returned))
    end-if
end-function

end-model
```

Related topics

[assign_var](#) [assign_and_forbid](#) [probe_assign_var](#) [split_domain](#)

cp_set_branching

Purpose

Sets the branching strategy to use during the search for a solution; please refer to the 'Paradigm' section of this reference manual for further details.

Synopsis

```
procedure cp_set_branching
procedure cp_set_branching(Strategy:cpbranching)
procedure cp_set_branching(Strategy:array(range) of cpbranching)
```

Argument

Strategy a branching strategy (single cpbranching or cpbranching array)

Example

The following example shows how to set the search strategy:

```
model "User branching"
uses "kalis"

parameters
  ALG=1
end-parameters

forward function varchoice(Vars: cpvarlist): integer
forward function varchoice2(Vars: cpvarlist): integer
forward function valchoice(x: cpvar): integer
forward function valchoice2(x: cpvar): integer

setparam("KALIS_DEFAULT_LB", 0);
setparam("KALIS_DEFAULT_UB", 20)

declarations
  R = 1..10
  y: array(R) of cpvar
  C: array(R) of integer
  Strategy: array(range) of cpbranching
end-declarations

C:: [4, 7, 2, 6, 9, 0,-1, 3, 8,-2]

all_different(y)
forall(i in R | isodd(i)) y(i) >= y(i+1) + 1
y(4) + y(1) = 13; y(8) <= 15; y(7) <> 5

! Definition of user branching strategies:
Strategy(1):= assign_and_forbid("varchoice2", "valchoice", y)
Strategy(2):= assign_var("varchoice", "valchoice", y)
Strategy(3):= split_domain("varchoice", "valchoice2", y, true, 2)
Strategy(4):= split_domain("varchoice2", "valchoice", y, false, 5)

! Select a branching strategy
cp_set_branching(Strategy(ALG))

if cp_find_next_sol then
  forall(i in R) write(getsol(y(i)), " ")
```

```

        writeln
    end-if

!-----
! **** Variable choice ****
! **** Choose variable with largest degree + smallest domain
function varchoice(Vars: cpvarlist): integer
    declarations
        Vset,Iset: set of integer
    end-declarations

    ! Get the number of elements of "Vars"
    listsize:= getsize(Vars)

    ! Set on uninstantiated variables
    forall(i in 1..listsize)
        if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

    if Vset={} then
        returned:= 0
    else
        ! Get the variables with max. degree
        dmax:= max(i in Vset) getdegree(getvar(Vars,i))
        forall(i in Vset)
            if getdegree(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
        dsize:= MAX_INT

        ! Choose var. with smallest domain among those indexed by 'Iset'
        forall(i in Iset)
            if getsize(getvar(Vars,i)) < dsize then
                returned:= i
                dsize:= getsize(getvar(Vars,i))
            end-if
        end-if
        writeln(returned)
    end-function

! **** Choose variable y(i) with smallest value of C(i)
function varchoice2(Vars: cpvarlist): integer
    declarations
        Vset,Iset: set of integer
        VarInd: array(Iset) of integer
    end-declarations

    ! Set on uninstantiated variables
    listsize:= getsize(Vars)
    forall(i in 1..listsize)
        if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

    if getsize(Vset)=0 then
        returned:= 0
    else
        ! Establish a correspondence of indices between 'Vars' and 'y'
        forall(i in R)

```

```

        forall(j in Vset)
        if is_same(getvar(Vars,j), y(i)) then
            VarInd(i):= j
            Vset -= {j}
            break 1
        end-if

! Choose the variable
imin:= min(i in Iset) i; cmin:= C(imin)
forall(i in Iset)
    if C(i) < cmin then
        imin:= i; cmin:= C(i)
    end-if
returned:= VarInd(imin)
end-if
writeln(imin, " ", returned)
end-function

!-----
! *** Value choice ****
! **** Choose the next value one third larger than lower bound
! (Strategy may be used with any branching scheme since it
!  makes sure that the chosen value lies in the domain)
function valchoice(x: cpvar): integer
!  returned:= getlb(x)
    returned:= getnext(x, getlb(x) + round((getub(x)-getlb(x))/3))
    writeln("Value: ", returned, " ", contains(x,returned),
            " x: ", x)
end-function

! **** Split the domain into lower third and upper two thirds
! (Strategy to be used only with 'split_domain' branching since
!  the chosen value may not be in the domain)
function valchoice2(x: cpvar): integer
    returned:= getlb(x) + round((getub(x)-getlb(x))/3)
    writeln("Value: ", returned, " x: ", x)
end-function
end-model

```

Related topics

[settle_disjunction](#) [assign_var](#) [assign_and_forbid](#) [split_domain](#) [Chapter 4](#)

cp_show_stats

Purpose

This procedure prints some statistics about the search process (time elapsed, number of nodes, depth, number of backtracks)

Synopsis

```
procedure cp_show_stats
```

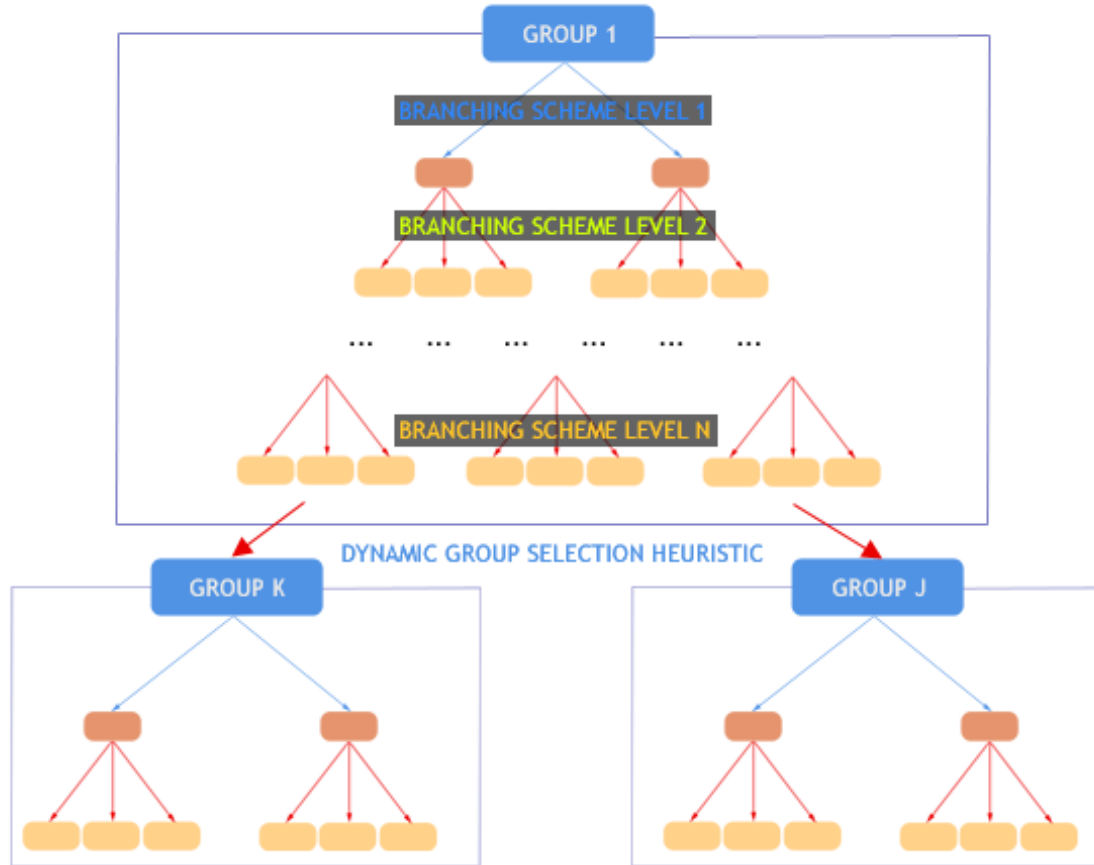
Related topics

[cp_show_prob](#) [cp_show_var](#)

bs_group

Purpose

Create a group of branching schemes. A group of branching schemes is an ordered list of branching schemes. It is used as input to the `bs_group_serializer` branching scheme. A user 'tag' information can be associated with a group in order to link this group to a specific user structure.



Synopsis

```
function bs_group(branchings: set of cpbranching, tag:integer) : cpbsgroup
function bs_group(branchings: array of cpbranching, tag:integer) : cpbsgroup
```

Arguments

branchings	the branching schemes forming the group
tag	tag associated with the branching scheme group (use and interpretation entirely up to the user)

Related topics

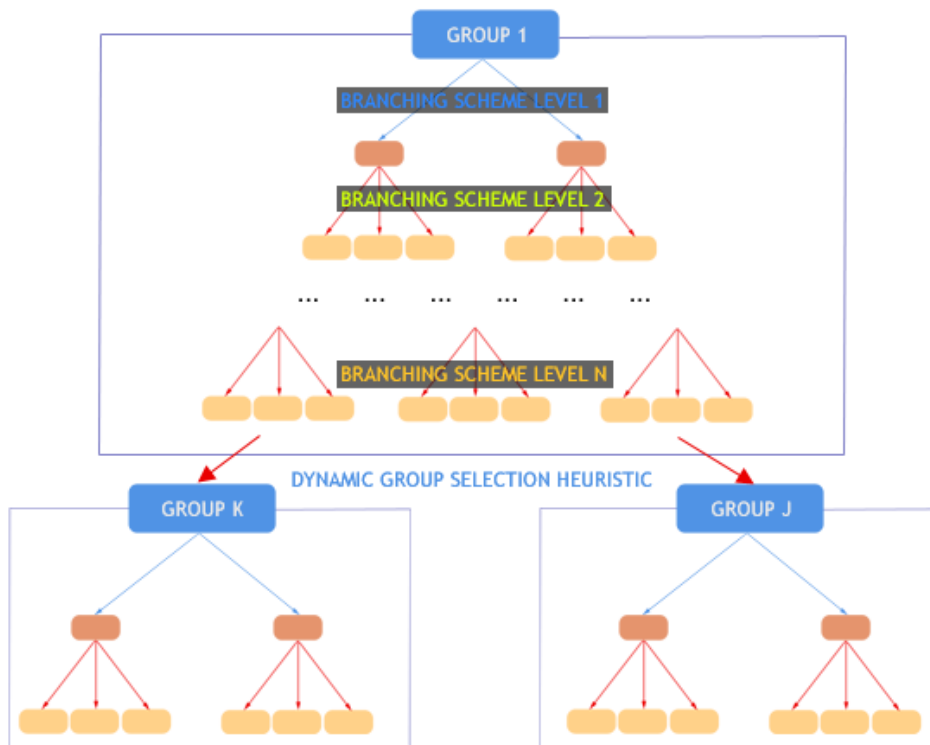
`group_serializer`

group_serializer

Purpose

A branching scheme Group Serializer is a branching scheme of branching schemes. That means, it can be seen as a meta-branching scheme. The search can be decomposed into clusters of branching schemes that will be explored based upon a specific dynamic group selection heuristic. When a specific group is selected, the ordered branching scheme list of this group will be treated sequentially. Once all the branching schemes of the group have been treated, another branching scheme group is selected with the group selection heuristic. For example, the task_serializer branching scheme is a particular kind of group serializer heuristic tailored for task based search tree exploration. In this case a group corresponds to a task and the list of branching schemes is the following :

1. start time variable
2. duration variable



Synopsis

```

function group_serializer(groups: set of cpbsgroup, groupSelector: string) :
    cpbranching
function group_serializer(groups: set of cpbsgroup, groupSelector: string, probe:
    integer) : cpbranching
  
```

Arguments

groups	the set of branching scheme groups
groupselector	the group selection heuristic
probe	the probe level

Related topics

[bs_group](#)

gettag

Purpose

Get the tag associated with the branching scheme group passed in argument. The tag is defined by the user when creating the group (see [bs_group](#)).

Synopsis

```
function gettag(group: cpbsgroup) : integer
```

Argument

group the set of branching scheme groups

Related topics

[group_serializer](#) [bs_group](#)

9.6 Callbacks

<code>cp_set_branch_callback</code>	Sets the branch callback procedure	p. 203
<code>cp_set_node_callback</code>	Sets the node callback procedure	p. 200
<code>cp_set_solution_callback</code>	Sets the solution callback procedure	p. 197

cp_set_solution_callback

Purpose

Sets the solution callback procedure that will be called each time that a solution has been found by the solver. Note that this callback procedure does not take any argument.

Synopsis

```
procedure cp_set_solution_callback(callbackName:string)
```

Argument

callbackName the name of the procedure to call

Example

The following example shows how to use the solution callback:

```
model "Using callbacks"
uses "kalis"

declarations
  NBTASKS = 5
  TASKS = 1..NBTASKS
  DUR: array(TASKS) of integer
  DURs: array(set of cpvar) of integer
  DUE: array(TASKS) of integer
  WEIGHT: array(TASKS) of integer
  start: array(TASKS) of cpvar
  tmp: array(TASKS) of cpvar
  tardiness: array(TASKS) of cpvar
  twt: cpvar
  zeroVar: cpvar
  Strategy: array(range) of cpbranching
  Disj: set of cpctr
  nodesx: array(range) of integer
  nodesy: array(range) of integer
  currentpos: integer
end-declarations

! Initialization of search tree data
nodesx(0) := 1
nodesy(0) := 0
currentpos := 1

! *****
! solution_found: called each time a solution is found
! *****
procedure solution_found
  writeln("A solution has been found :")
  forall (t in TASKS)
    writeln("[", getsol(start(t)), "==>",
            (getsol(start(t)) + DUR(t)), "]:\t ",
            getsol(tardiness(t)), "  (" , getsol(tmp(t)), ")")
  writeln("Total weighted tardiness: ", getsol(twt))
end-procedure

! *****
! node_explored: called each time a node is explored
```

```

! *****
procedure node_explored
  nodesx(currentpos) += 1
  if (nodesx(currentpos-1)>nodesx(currentpos)) then
    nodesx(currentpos) := nodesx(currentpos-1)
  end-if
  nodesy(currentpos) := -currentpos
  writeln("Node explored depth : " , (-nodesy(currentpos)), "]")
end-procedure

! *****
! go_down_branch: called each time the search goes down
!               a branch of the search tree
! *****
procedure go_down_branch
  writeln("Branch go_down " , (-nodesy(currentpos)) , "]")
  currentpos := currentpos + 1
end-procedure

! *****
! go_up_branch: called each time the search goes up
!               a branch of the search tree
! *****
procedure go_up_branch
  currentpos := currentpos - 1
  writeln("Branch go_up " , (-nodesy(currentpos)) , "]")
end-procedure

! *****
! Problem definition
! *****

DUR :: [21,53,95,55,34]
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

! Setting up the decision variables
forall (t in TASKS) do
  start(t) >= 0
  setname(start(t), "Start("+t+")")
  DURs(start(t)):= DUR(t)
  tmp(t) = start(t) + DUR(t) - DUE(t)
  setname(tardiness(t), "Tard("+t+")")
  tardiness(t) = maximum({tmp(t), zeroVar})
end-do

twt = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
disjunctive(union(t in TASKS) {start(t)}, DURs, Disj, 1)

```

```
! The setxxxxcallback methods must be called before
! setting the branching with 'cp_set_branching'
cp_set_solution_callback("solution_found")
cp_set_node_callback("node_explored")
cp_set_branch_callback("go_down_branch", "go_up_branch")

Strategy(0) := settle_disjunction
Strategy(1) := split_domain(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

if not(cp_minimize(twt)) then
    writeln("problem is inconsistent")
    exit(0)
end-if

end-model
```

Related topics

[getsol](#) [cp_set_node_callback](#) [cp_set_branch_callback](#)

cp_set_node_callback

Purpose

Sets the node callback procedure that will be called each time that a node is explored by the search algorithm. Note that this callback procedure does not take any argument.

Synopsis

```
procedure cp_set_node_callback(callbackName:string)
```

Argument

callbackName the name of the procedure to call

Example

The following example shows how to use the node callback:

```
model "Using callbacks"
uses "kalis"

declarations
  NBTASKS = 5
  TASKS = 1..NBTASKS                                ! Set of tasks
  DUR: array(TASKS) of integer                       ! Task durations
  DURs: array(set of cpvar) of integer               ! Durations
  DUE: array(TASKS) of integer                       ! Due dates
  WEIGHT: array(TASKS) of integer                   ! Weights of tasks
  start: array(TASKS) of cpvar                      ! Start times
  tmp: array(TASKS) of cpvar                        ! Aux. variable
  tardiness: array(TASKS) of cpvar                  ! Tardiness
  twt: cpvar                                         ! Objective variable
  zeroVar: cpvar                                    ! 0-valued variable
  Strategy: array(range) of cpbranching             ! Branching strategy
  Disj: set of cpctr                                ! Disjunctions
  nodesx: array(range) of integer                   ! x-coordinates
  nodesy: array(range) of integer                   ! y-coordinates
  currentpos: integer
end-declarations

! Initialization of search tree data
nodesx(0) := 1
nodesy(0) := 0
currentpos := 1

! *****
! solution_found: called each time a solution is found
! *****
procedure solution_found
  writeln("A solution has been found :")
  forall (t in TASKS)
    writeln("[", getsol(start(t)), "==>",
            (getsol(start(t)) + DUR(t)), "]:\t ",
            getsol(tardiness(t)), "  (" , getsol(tmp(t)), ")")
  writeln("Total weighted tardiness: ", getsol(twt))
end-procedure

! *****
! node_explored: called each time a node is explored
```

```

! *****
procedure node_explored
  nodesx(currentpos) += 1
  if (nodesx(currentpos-1)>nodesx(currentpos)) then
    nodesx(currentpos) := nodesx(currentpos-1)
  end-if
  nodesy(currentpos) := -currentpos
  writeln("Node explored depth : " , (-nodesy(currentpos)), "]")
end-procedure

! *****
! go_down_branch: called each time the search goes down
!                   a branch of the search tree
! *****
procedure go_down_branch
  writeln("Branch go_down " , (-nodesy(currentpos)) , "]")
  currentpos := currentpos + 1
end-procedure

! *****
! go_up_branch: called each time the search goes up
!                   a branch of the search tree
! *****
procedure go_up_branch
  currentpos := currentpos - 1
  writeln("Branch go_up " , (-nodesy(currentpos)) , "]")
end-procedure

! *****
! Problem definition
! *****

DUR :: [21,53,95,55,34]
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

! Setting up the decision variables
forall (t in TASKS) do
  start(t) >= 0
  setname(start(t), "Start("+t+")")
  DURs(start(t)):= DUR(t)
  tmp(t) = start(t) + DUR(t) - DUE(t)
  setname(tardiness(t), "Tard("+t+")")
  tardiness(t) = maximum({tmp(t), zeroVar})
end-do

twt = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
disjunctive(union(t in TASKS) {start(t)}, DURs, Disj, 1)

```

```
! The setxxxxcallback methods must be called before
! setting the branching with 'cp_set_branching'
cp_set_solution_callback("solution_found")
cp_set_node_callback("node_explored")
cp_set_branch_callback("go_down_branch", "go_up_branch")

Strategy(0) := settle_disjunction
Strategy(1) := split_domain(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

if not(cp_minimize(twt)) then
    writeln("problem is inconsistent")
    exit(0)
end-if

end-model
```

Related topics

[getsol](#) [cp_set_solution_callback](#) [cp_set_branch_callback](#)

cp_set_branch_callback

Purpose

Sets the branch callback procedure that will be called each time that a branch is explored by the search algorithm. Note that this callback procedure does not take any argument.

Synopsis

```
procedure cp_set_branch_callback(callbackNameDown:string, callbackNameUp:string)
```

Argument

callbackNameDown, callbackNameUp	the name of the procedure to call, in the case the branch is explored downward (resp. upward)
----------------------------------	---

Example

The following example shows how to use the branch callback:

```
model "Using callbacks"
uses "kalis"

declarations
  NBTASKS = 5
  TASKS = 1..NBTASKS
  DUR: array(TASKS) of integer
  DURs: array(set of cpvar) of integer
  DUE: array(TASKS) of integer
  WEIGHT: array(TASKS) of integer
  start: array(TASKS) of cpvar
  tmp: array(TASKS) of cpvar
  tardiness: array(TASKS) of cpvar
  twt: cpvar
  zeroVar: cpvar
  Strategy: array(range) of cpbranching
  Disj: set of cpctr
  nodesx: array(range) of integer
  nodesy: array(range) of integer
  currentpos: integer
end-declarations

! Initialization of search tree data
nodesx(0) := 1
nodesy(0) := 0
currentpos := 1

! *****
! solution_found: called each time a solution is found
! *****
procedure solution_found
  writeln("A solution has been found :")
  forall (t in TASKS)
    writeln("[", getsol(start(t)), "==>",
            (getsol(start(t)) + DUR(t)), "]:\t ",
            getsol(tardiness(t)), "  (" , getsol(tmp(t)), ")")
  writeln("Total weighted tardiness: ", getsol(twt))
end-procedure

! *****
```

```

! node_explored: called each time a node is explored
! *****
procedure node_explored
  nodesx(currentpos) += 1
  if (nodesx(currentpos-1)>nodesx(currentpos)) then
    nodesx(currentpos) := nodesx(currentpos-1)
  end-if
  nodesy(currentpos) := -currentpos
  writeln("Node explored depth : " , (-nodesy(currentpos)), "]")
end-procedure

! *****
! go_down_branch: called each time the search goes down
!               a branch of the search tree
! *****
procedure go_down_branch
  writeln("Branch go_down " , (-nodesy(currentpos)) , "]")
  currentpos := currentpos + 1
end-procedure

! *****
! go_up_branch: called each time the search goes up
!               a branch of the search tree
! *****
procedure go_up_branch
  currentpos := currentpos - 1
  writeln("Branch go_up " , (-nodesy(currentpos)) , "]")
end-procedure

! *****
! Problem definition
! *****

DUR :: [21,53,95,55,34]
DUE :: [66,101,232,125,150]
WEIGHT :: [1,1,1,1,1]

setname(twt, "Total weighted tardiness")
zeroVar = 0
setname(zeroVar, "zeroVar")

! Setting up the decision variables
forall (t in TASKS) do
  start(t) >= 0
  setname(start(t), "Start("+t+")")
  DURs(start(t)):= DUR(t)
  tmp(t) = start(t) + DUR(t) - DUE(t)
  setname(tardiness(t), "Tard("+t+")")
  tardiness(t) = maximum({tmp(t), zeroVar})
end-do

twt = sum(t in TASKS) (WEIGHT(t) * tardiness(t))

! Create the disjunctive constraints
disjunctive(union(t in TASKS) {start(t)}, DURs, Disj, 1)

```

```
! The setxxxxcallback methods must be called before
! setting the branching with 'cp_set_branching'
cp_set_solution_callback("solution_found")
cp_set_node_callback("node_explored")
cp_set_branch_callback("go_down_branch", "go_up_branch")

Strategy(0) := settle_disjunction
Strategy(1) := split_domain(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

if not(cp_minimize(twt)) then
    writeln("problem is inconsistent")
    exit(0)
end-if

end-model
```

Related topics

[getsol](#) [cp_set_solution_callback](#) [cp_set_node_callback](#)

9.7 Scheduling

<code>addpredecessors</code>	Adds a set of predecessors for a task	p. 211
<code>addsuccessors</code>	Adds a set of tasks as successors of a task	p. 213
<code>consumes</code>	Sets the minimal and maximal amount of resource consumed by a task for a particular resource	p. 233
<code>cp_close_schedule</code>	Close the schedule.	p. 268
<code>cp_get_default_schedule_strategy</code>	Gets the default schedule search strategy of <code>cp_schedule</code>	p. 258
<code>cp_schedule</code>	Optimizes the schedule with respect to an objective variable.	p. 231
<code>cp_set_schedule_strategy</code>	Sets the schedule search strategy for <code>cp_schedule</code>	p. 257
<code>cp_show_schedule</code>	Shows a textual representation of the current schedule	p. 215
<code>getassignment</code>	Gets the cpvar representing the assignment of a task for a particular resource.	p. 262
<code>getcapacity</code>	Get the maximal capacity of a resource for a specific time period.	p. 252
<code>getconsumption</code>	Gets the cpvar representing the consumption of a task for a particular resource	p. 219
<code>getduration</code>	Gets the cpvar representing a task duration	p. 218
<code>getend</code>	Gets the cpvar representing a task completion time	p. 217
<code>getmakespan</code>	Gets the cpvar representing the makespan of the schedule.	p. 232
<code>getproduction</code>	Gets the cpvar representing the production of a task for a particular resource	p. 223
<code>getprovision</code>	Gets the cpvar representing the provision of a task for a particular resource	p. 225
<code>getrequirement</code>	Gets the cpvar representing the requirement of a task for a particular resource	p. 221
<code>getsetuptime</code>	Gets the sequence dependent setup times between two tasks	p. 255
<code>getstart</code>	Gets the cpvar representing a task start time	p. 216
<code>has_assignment</code>	Tests whether an assignment decision variable for a task and a particular resource exists.	p. 264
<code>is_consuming</code>	Tests whether a task consumes a specific resource	p. 227
<code>is_fixed</code>	Tests if a task is fixed	p. 208
<code>is_fixed</code>	Tests if a disjunction is fixed	p. 256
<code>is_idletime</code>	Tests if a timestep is an idle timestep for a resource.	p. 267
<code>is_producing</code>	Tests whether a task produces a specific resource	p. 229
<code>is_providing</code>	Tests whether a task provides a specific resource	p. 230

<code>is_requiring</code>	Tests whether a task requires a specific resource	p. 228
<code>produces</code>	Sets the minimal and maximal amount of resource produced by a task for a particular resource	p. 239
<code>provides</code>	Sets the minimal and maximal amount of resource provided by a task for a particular resource.	p. 242
<code>requires</code>	Sets the minimal and maximal amount of resource required by a task for a particular resource	p. 236
<code>resusage</code>	Creates a resource usage	p. 259
<code>set_resource_attributes</code>	Sets some attributes for a resource	p. 245
<code>set_task_attributes</code>	Sets some attributes for a task	p. 209
<code>setcapacity</code>	Sets the maximal capacity of a resource between two time bounds. p. 246	
<code>setduration</code>	Sets the duration of a task	p. 210
<code>setidletimes</code>	Specifies the set of timesteps where a resource is idle.	p. 266
<code>setmaxavailability</code>	Sets the maximal capacity of a resource between two time bounds. p. 248	
<code>setminusage</code>	Sets the minimum usage of a resource between two time bounds. p. 250	
<code>setpredecessors</code>	Sets the tasks that must precede a task	p. 212
<code>setsetuptime</code>	Sets sequence dependent setup times between two tasks	p. 254
<code>setsuccessors</code>	Sets the set of tasks that must succeed a task	p. 214

is_fixed

Purpose

Returns `true` if the task passed in argument has its start and completion times fixed.

Synopsis

```
function is_fixed(task:cptask) : boolean
```

Argument

`task` the task

Return value

`true` if task is fixed

Example

The following example shows how to see if a `cptask` task is fixed

```
if is_fixed(task) then
  write('task is fixed! ')
end-if
```

Related topics

[getlb](#) [getub](#) [getmiddle](#) [getsize](#) [getval](#) [getdegree](#) [gettarget](#) [getrand](#) [getnext](#) [getprev](#) [contains](#)

set_task_attributes

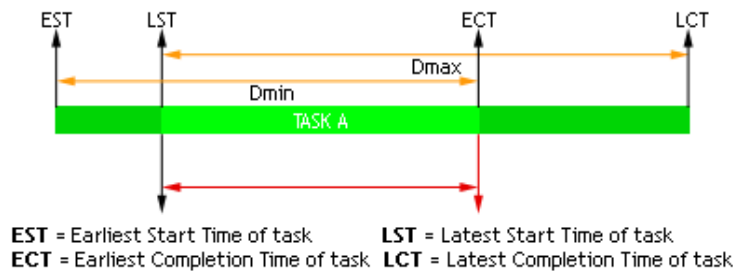
Purpose

Sets some attributes for a task.

A cptask is represented by three cpvar:

- start representing the start time of the task
- end representing the completion time of the task
- duration representing the duration of the task.

These three structural variables are linked with the following constraint: $\text{task} + \text{duration} \leq \text{end}$. The start time variable represents two specific parameters of the task: the Earliest Start Time (EST, represented by its lower bound) and its Latest Start Time (LST, represented by its upper bound). The end variable represents another two parameters of the task: the Earliest Completion Time (ECT, represented by its lower bound) and its Latest Completion Time (LCT, represented by its upper bound). The duration variable represents the following two parameters of the task: the minimum task duration (Dmin, represented by its lower bound) and the maximum task duration (Dmax, represented by its upper bound). The graphic below illustrates these properties:



Synopsis

```
procedure set_task_attributes(task:cptask, duration:integer, resource:cpresource,
                             precedences:set of cptask)
procedure set_task_attributes(task:cptask, duration:integer, resource:cpresource)
procedure set_task_attributes(task:cptask, duration:integer)
procedure set_task_attributes(task:cptask, duration:integer, precedences:set of
                             cptask)
procedure set_task_attributes(task:cptask, resource:cpresource, requirement:integer)
```

Arguments

task	the task to set attributes
duration	the duration of the task
resource	a resource that is required during the execution of the task
precedences	the set of tasks that must precede t in the schedule
requirement	amount of the specified resource required per time unit

Example

The following example shows how to set some attributes of a task:

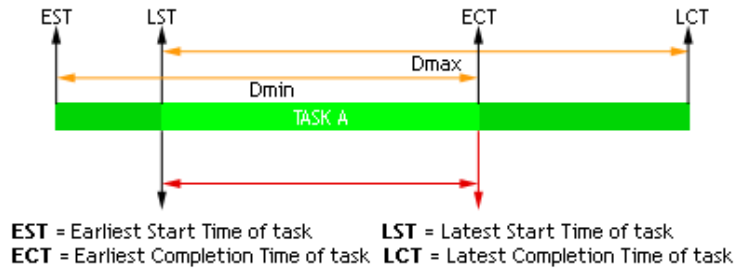
setduration

Purpose

A cptask is represented by three cpvar:

- 'start' representing the start time of the task
- 'end' representing the completion time of the task
- 'duration' representing the duration of the task.

These three structural variables are linked with the following constraint: $\text{task} + \text{duration} \leq \text{end}$. The duration variable represents two specific parameters of the task: The minimum task duration (Dmin, represented by its lower bound) and the maximum task duration (Dmax, represented by its upper bound). The graphic below illustrates these properties:



Synopsis

```
procedure setduration(task:cptask, duration:integer)
procedure setduration(task:cptask, durationMin:integer, durationMax:integer)
```

Arguments

task	a task
duration	fixed duration of the task
durationMin	minimum duration for the task
durationMax	maximum duration for the task

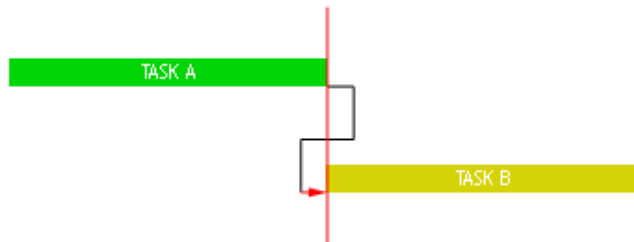
Related topics

[getduration](#)

addpredecessors

Purpose

Adds a set of tasks as predecessors to a task. A task precedes some other task when it must be completed before the start of the second task. The following graphic shows two tasks A and B where A precedes B (and respectively B succeeds A as the relation is reflexive):



Synopsis

```
procedure addpredecessors(task:cptask, predset:set of cptask)
```

Arguments

task	the task
predset	the set of tasks that must precede 'task'

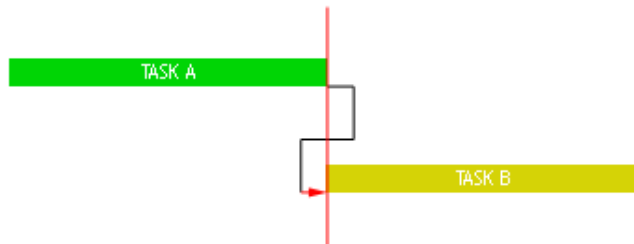
Related topics

[addsucessors](#) [setpredecessors](#) [setsucessors](#)

setpredecessors

Purpose

Sets the set of tasks that must precede a task. A task precedes some other task when it must be completed before the start of the second task. The following graphic shows two tasks A and B where A precedes B (and respectively B succeeds A as the relation is reflexive):



Synopsis

```
procedure setpredecessors(task:cptask, predset:set of cptask)
```

Arguments

task	the task
predset	the set of tasks that must precede 'task'

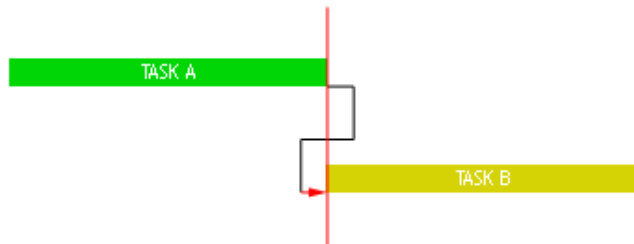
Related topics

[addpredecessors](#) [addsucessors](#) [setsuccessors](#)

addsuccessors

Purpose

Adds a set of tasks as successors of a task. A task succeeds some other task when its processing cannot start before the completion of this task. The following graphic shows two tasks A and B where A precedes B (and respectively B succeeds to A as the relation is reflexive):



Synopsis

```
procedure addsuccessors(task:cptask, succset:set of cptask)
```

Arguments

task	the task
succset	the set of tasks that must succeed 'task'

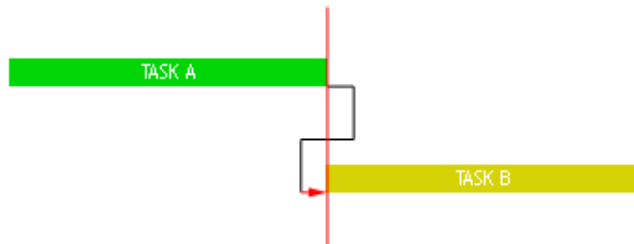
Related topics

[addpredecessors](#) [setpredecessors](#) [setsuccessors](#)

setsuccessors

Purpose

Sets the set of tasks that must succeed a task. A task succeeds some other task when its processing cannot start before the completion of this task. The following schema shows two tasks A and B where A precedes B (and respectively B succeeds A as the relation is reflexive):



Synopsis

```
procedure setsuccessors(task:cptask, succset:set of cptask)
```

Arguments

task	the task
succset	the set of tasks that must succeed 'task'

Related topics

[addpredecessors](#) [addsuccors](#) [setpredecessors](#)

cp_show_schedule

Purpose

Shows a textual representation of the current schedule

Synopsis

```
procedure cp_show_schedule
```


getstart

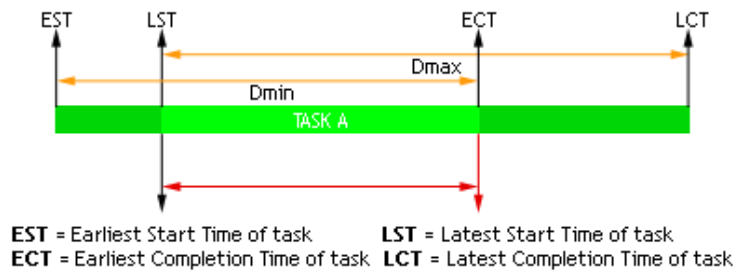
Purpose

Retrieves the 'start' variable of a task.

A cptask is represented by three cpvar:

- 'start' representing the start time of the task
- 'end' representing the completion time of the task
- 'duration' representing the duration of the task.

These three structural variables are linked with the following constraint: $\text{start} + \text{duration} \leq \text{end}$. The start time variable represents two specific parameters of the task: the Earliest Starting Time (EST represented by its lower bound) and its Latest Starting Time (LST represented by its upper bound). The graphic below illustrates these properties:



Synopsis

```
function getstart(t:cptask) : cpvar
```

Argument

t the task

Return value

The cpvar representing the start time of t

Related topics

[getend](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getproduction](#) [getprovision](#)

getend

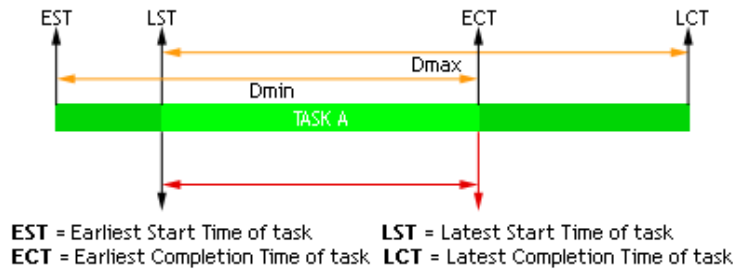
Purpose

Retrieves the 'end' variable of a task.

A cptask is represented by three cpvar:

- 'start' representing the start time of the task
- 'end' representing the completion time of the task
- 'duration' representing the duration of the task.

These three structural variables are linked with the following constraint: $\text{start} + \text{duration} \leq \text{end}$. The 'end' variable represents two specific parameters of the task: the Earliest Completion Time (ECT, represented by its lower bound) and its Latest Completion Time (LCT, represented by its upper bound). The graphic below illustrates these properties:



Synopsis

```
function getend(t:cptask) : cpvar
```

Argument

t the task

Return value

The cpvar representing the completion time of t

Related topics

[getstart](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getproduction](#) [getprovision](#)

getduration

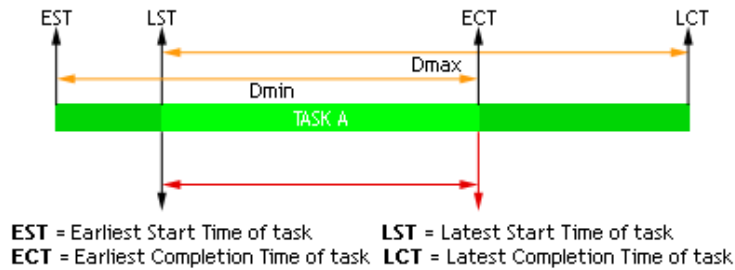
Purpose

Retrieves the 'duration' variable of a task.

A cptask is represented by three cpvar:

- 'start' representing the start time of the task
- 'end' representing the completion time of the task
- 'duration' representing the duration of the task.

These three structural variables are linked with the following constraint: $\text{start} + \text{duration} \leq \text{end}$. The duration variable represents two specific parameters of the task: the minimum task duration (Dmin, represented by its lower bound) and the maximum task duration (Dmax represented by its upper bound). The graphic below illustrates these properties:



Synopsis

```
function getduration(t:cptask) : cpvar
```

Argument

t the task

Return value

The cpvar representing the duration of t

Related topics

[getstart](#) [getend](#) [getconsumption](#) [getrequirement](#) [getproduction](#) [getprovision](#)

getconsumption

Purpose

Gets the cpvar representing the consumption of a task for a particular resource.
A task consumes some amount of processing power of a resource when this amount must be made available for the execution of the task. The resource requirement is considered as renewable which means that whenever the task ends the processing power that was used is no longer available for processing other tasks.

Synopsis

```
function getconsumption(task:cptask, resource:cpresource) : cpvar
function getconsumption(task:cptask, resource:cpresource, p:integer) : integer
```

Arguments

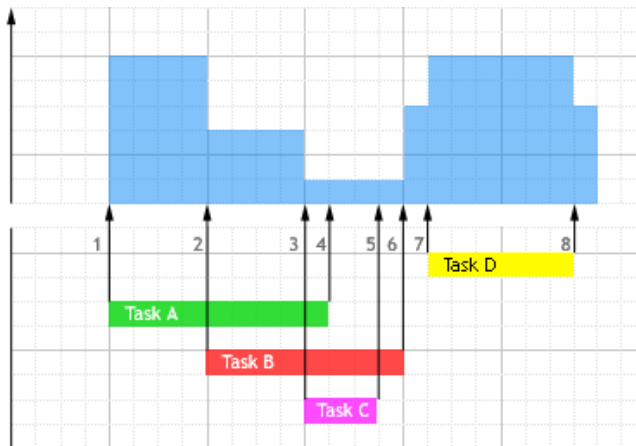
task	the task
resource	the resource
p	the timestep

Return value

The cpvar representing the consumption of 'resource' by 'task'

Example

The following example illustrates this:



Task A produces 6 units of resource

Task B requires 3 units of resource

Task C consumes 2 units of resource

Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```
model "Producer Consumer"
uses "kalis"
```

```
declarations
```

```
Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
Ceiling, Painting, MovingIn : cptask    ! Declaration of tasks
```

```
    money_available : cresource          ! Resource declaration
end-declarations

! 'money_available' is a cumulative resource with max. amount of 29$
set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

! Limit resource availability to 20$ in the time interval [0,14]
setcapacity(money_available, 0, 14, 20)

! Setting the task durations and predecessor sets
set_task_attributes(Masonry , 7 )
set_task_attributes(Carpentry, 3, {Masonry} )
set_task_attributes(Roofing , 1, {Carpentry} )
set_task_attributes(Windows , 1, {Roofing} )
set_task_attributes(Facade , 2, {Roofing} )
set_task_attributes(Garden , 1, {Roofing} )
set_task_attributes(Plumbing , 8, {Masonry} )
set_task_attributes(Ceiling , 3, {Masonry} )
set_task_attributes(Painting , 2, {Ceiling} )
set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

! Setting the resource consumptions
consumes(Masonry , 7, money_available )
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[getstart](#) [getend](#) [getduration](#) [getrequirement](#) [getproduction](#) [getprovision](#)

getrequirement

Purpose

Gets the cpvar representing the requirement of a task for a particular resource.
A task requires some amount of processing power of a resource when this amount must be made available for the execution of the task. The resource requirement is considered as recoverable which means that whenever the task ends the processing power that was used by it becomes available again for processing other tasks.

Synopsis

```
function getrequirement(task:cptask, resource:cpresource) : cpvar
function getrequirement(task:cptask, resource:cpresource, p:integer) : integer
```

Arguments

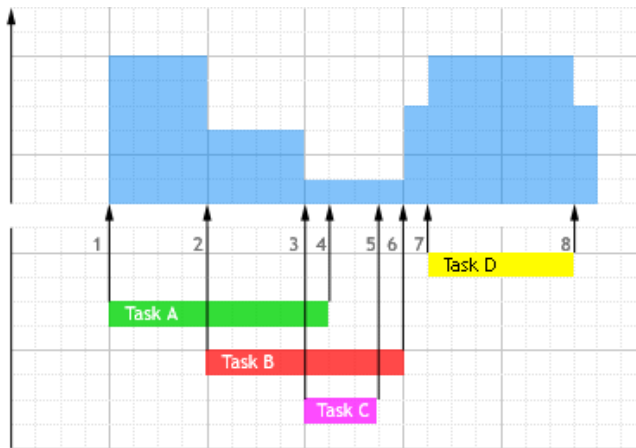
task	the task
resource	the resource
p	the timestep

Return value

The cpvar representing the requirement of 'resource' by 'task'

Example

The following example illustrates this:



Task A produces 6 units of resource

Task B requires 3 units of resource

Task C consumes 2 units of resource

Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```
model "Producer Consumer"
uses "kalis"
```

```
declarations
```

```
Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
Ceiling, Painting, MovingIn : cptask    ! Declaration of tasks
```

```
    money_available : cresource           ! Resource declaration
end-declarations

! 'money_available' is a cumulative resource with max. amount of 29$
set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

! Limit resource availability to 20$ in the time interval [0,14]
setcapacity(money_available, 0, 14, 20)

! Setting the task durations and predecessor sets
set_task_attributes(Masonry , 7 )
set_task_attributes(Carpentry, 3, {Masonry} )
set_task_attributes(Roofing , 1, {Carpentry} )
set_task_attributes(Windows , 1, {Roofing} )
set_task_attributes(Facade , 2, {Roofing} )
set_task_attributes(Garden , 1, {Roofing} )
set_task_attributes(Plumbing , 8, {Masonry} )
set_task_attributes(Ceiling , 3, {Masonry} )
set_task_attributes(Painting , 2, {Ceiling} )
set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

! Setting the resource consumptions
consumes(Masonry , 7, money_available )
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[getstart](#) [getend](#) [getduration](#) [getconsumption](#) [getproduction](#) [getprovision](#)

getproduction

Purpose

Gets the cpvar representing the production of a task for a particular resource.
A task produces some amount of processing power for a resource at its start. The capacity produced is considered as non-renewable which means that the produced capacity remains available even after the termination of the task.

Synopsis

```
function getproduction(task:cptask, resource:cpresource) : cpvar
function getproduction(task:cptask, resource:cpresource, p:integer) : integer
```

Arguments

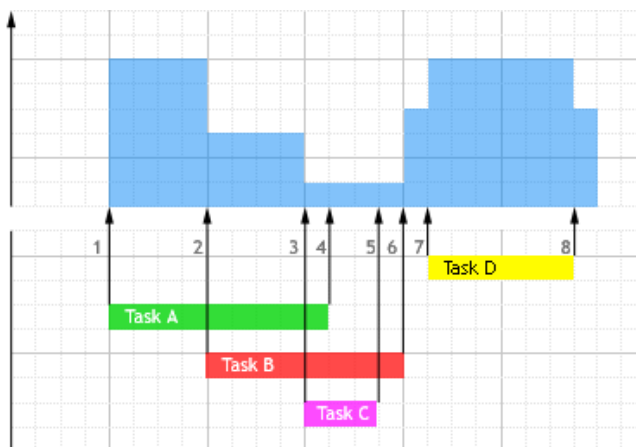
task	the task
resource	the resource
p	the timestep

Return value

The cpvar representing the production of 'resource' by 'task'

Example

The following example illustrates this:



Task A produces 6 units of resource
Task B requires 3 units of resource
Task C consumes 2 units of resource
Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```
model "Producer Consumer"
uses "kalis"

declarations
  Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
    Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
  money_available : cresource                 ! Resource declaration
```



```
end-declarations

! 'money_available' is a cumulative resource with max. amount of 29$
set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

! Limit resource availability to 20$ in the time interval [0,14]
setcapacity(money_available, 0, 14, 20)

! Setting the task durations and predecessor sets
set_task_attributes(Masonry , 7 )
set_task_attributes(Carpentry, 3, {Masonry} )
set_task_attributes(Roofing , 1, {Carpentry} )
set_task_attributes(Windows , 1, {Roofing} )
set_task_attributes(Facade , 2, {Roofing} )
set_task_attributes(Garden , 1, {Roofing} )
set_task_attributes(Plumbing , 8, {Masonry} )
set_task_attributes(Ceiling , 3, {Masonry} )
set_task_attributes(Painting , 2, {Ceiling} )
set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

! Setting the resource consumptions
consumes(Masonry , 7, money_available )
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[getstart](#) [getend](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getprovision](#)

getprovision

Purpose

Gets the cpvar representing the provision of a task for a particular resource.
A task provides some amount of processing power for a resource during its execution. The capacity provided is considered as renewable which means that whenever the providing task ends, the provided capacity is no longer available.

Synopsis

```
function getprovision(task:cptask, resource:cpresource) : cpvar
function getprovision(task:cptask, resource:cpresource, p:integer) : integer
```

Arguments

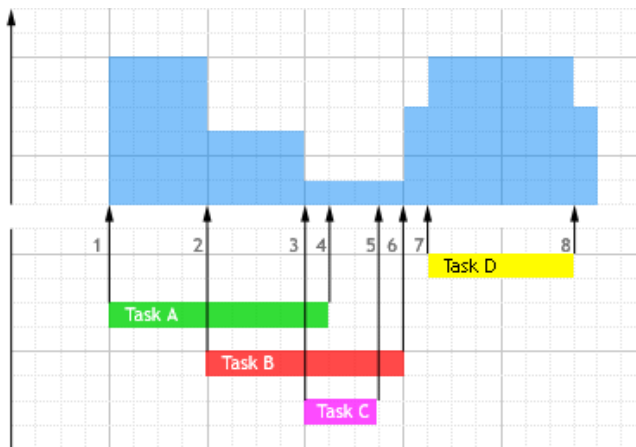
task	the task
resource	the resource
p	the timestep

Return value

The cpvar representing the provision of 'resource' by 'task'

Example

The following example illustrates this:



Task A produces 6 units of resource

Task B requires 3 units of resource

Task C consumes 2 units of resource

Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drops down of two units as the resource provision is recoverable.

```
model "Producer Consumer"
uses "kalis"
```

```
declarations
```

```
  Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
    Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
  money_available : cresource                ! Resource declaration
```

```
end-declarations

! 'money_available' is a cumulative resource with max. amount of 29$
set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

! Limit resource availability to 20$ in the time interval [0,14]
setcapacity(money_available, 0, 14, 20)

! Setting the task durations and predecessor sets
set_task_attributes(Masonry , 7 )
set_task_attributes(Carpentry, 3, {Masonry} )
set_task_attributes(Roofing , 1, {Carpentry} )
set_task_attributes(Windows , 1, {Roofing} )
set_task_attributes(Facade , 2, {Roofing} )
set_task_attributes(Garden , 1, {Roofing} )
set_task_attributes(Plumbing , 8, {Masonry} )
set_task_attributes(Ceiling , 3, {Masonry} )
set_task_attributes(Painting , 2, {Ceiling} )
set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

! Setting the resource consumptions
consumes(Masonry , 7, money_available )
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[getstart](#) [getend](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getproduction](#)

is_consuming

Purpose

Returns true IFF the specified task consumes a specific resource.

Synopsis

```
function is_consuming(task:cptask, resource:cpresource) : boolean
```

Arguments

task	the task
resource	the resource

Return value

true IFF the task consumes the specified resource, false otherwise

Related topics

[is_requiring](#) [is_producing](#) [is_providing](#)

is_requiring

Purpose

Returns true IFF the specified task requires a specific resource.

Synopsis

```
function is_requiring(task:cptask, resource:cpresource) : boolean
```

Arguments

task	the task
resource	the resource

Return value

true IFF the task requires the specified resource, false otherwise

Related topics

[is_consuming](#) [is_producing](#) [is_providing](#)

is_producing

Purpose

Returns true IFF the specified task produces a specific resource.

Synopsis

```
function is_producing(task:cptask, resource:cpresource) : boolean
```

Arguments

task	the task
resource	the resource

Return value

true IFF the task produces the specified resource, false otherwise

Related topics

[is_consuming](#) [is_requiring](#) [is_providing](#)

is_providing

Purpose

Returns true IFF the specified task provides a specific resource.

Synopsis

```
function is_providing(task:cptask, resource:cpresource) : boolean
```

Arguments

task	the task
resource	the resource

Return value

true IFF the task provides the specified resource, false otherwise

Related topics

[is_consuming](#) [is_requiring](#) [is_producing](#)

cp_schedule

Purpose

Optimizes the schedule with respect to an objective variable.

Synopsis

```
function cp_schedule(obj:cpvar) : integer
function cp_schedule(obj:cpfloatvar) : integer
function cp_schedule(obj:cpfloatvar, maximization:boolean) : integer
function cp_schedule(obj:cpvar, maximization:boolean) : integer
```

Arguments

obj	the objective variable
maximization	true for maximization of the objective and false for minimization (default: minimization)

Return value

0 if schedule is inconsistent, 1 if schedule is suboptimal, 2 if schedule is optimal

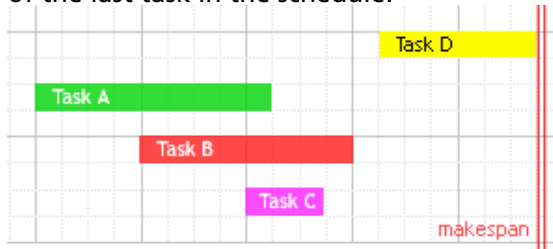
Related topics

[getmakespan](#) [cp_set_schedule_strategy](#)

getmakespan

Purpose

Gets the cpvar representing the makespan of the schedule. The makespan is the completion time of the last task in the schedule.



Synopsis

```
function getmakespan : cpvar
```

Return value

The cpvar representing the makespan of the schedule

Related topics

[cp_schedule](#)

consumes

Purpose

Sets the minimal and maximal amount of resource consumed by a task for a particular resource. A task consumes some amount of processing power of a resource when this amount must be made available for the execution of the task. The resource requirement is considered as renewable which means that whenever the task ends the processing power that was required is no longer available for processing other tasks. Note that a minimal amount of 0 is possible thus allowing modeling of alternative resources. The type of resource consumption can be defined in several ways:

- By specifying a resource and a constant consumption parameter '*consumption*'. In this case the resource consumption is constant other the execution of the task.
- By specifying a resource and a minimal '*minCons*' and maximal '*maxCons*' consumption parameters: In this case the resource consumption is a decision variable but is constant other the execution of the task.
- By specifying a cpresusage '*usage*' (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*': This makes it possible to model alternative resources. The type of resource consumption is defined by the resource usage passed in argument (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*', a minimal '*minalt*' and a maximal '*maxalt*' number of alternative resources that must be consumed: Between [*minalt*..*maxalt*] alternative consumptions from the set defined will be active which means that several resources will be chosen to be consumed by the task. The type of resource consumption is defined by the resource usage passed in argument (see [resusage](#) for further detail).

Synopsis

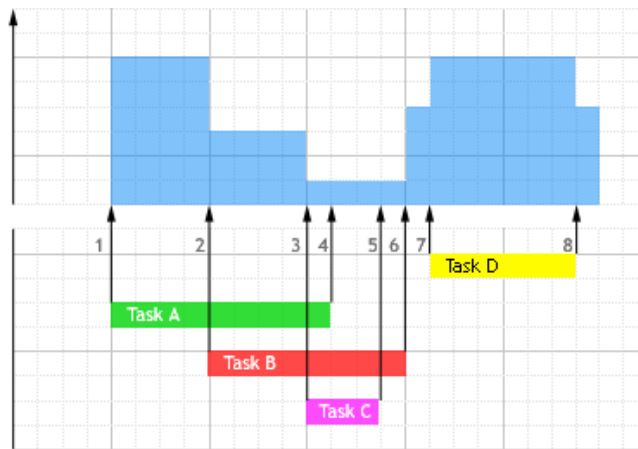
```
procedure consumes(task:cptask, consumption:integer, resource:cpresource)
procedure consumes(task:cptask, consumptionMin:integer, consumptionMax,
    resource:cpresource)
procedure consumes(task:cptask, usage:cpresusage)
procedure consumes(task:cptask, optUsages:set of cpresusage)
procedure consumes(task:cptask, optUsages:set of cpresusage, minalt:integer,
    maxalt:integer)
```

Arguments

task	the task
consumption	constant positive resource consumption
minCons	minimal resource consumption
maxCons	maximal resource consumption
resource	the resource to be consumed
usage	the resource usage
optUsages	the set of alternatives resource consumption
minalt	minimal number of resources that must be consumed
maxalt	maximal number of resources that may be consumed

Example

The following example illustrates this:



Task A produces 6 units of resource
 Task B requires 3 units of resource
 Task C consumes 2 units of resource
 Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```

model "Producer Consumer"
  uses "kalis"

  declarations
    Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
      Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
    money_available : cpresource                ! Resource declaration
  end-declarations

  ! 'money_available' is a cumulative resource with max. amount of 29$
  set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

  ! Limit resource availability to 20$ in the time interval [0,14]
  setcapacity(money_available, 0, 14, 20)

  ! Setting the task durations and predecessor sets
  set_task_attributes(Masonry , 7 )
  set_task_attributes(Carpentry, 3, {Masonry} )
  set_task_attributes(Roofing , 1, {Carpentry} )
  set_task_attributes(Windows , 1, {Roofing} )
  set_task_attributes(Facade , 2, {Roofing} )
  set_task_attributes(Garden , 1, {Roofing} )
  set_task_attributes(Plumbing , 8, {Masonry} )
  set_task_attributes(Ceiling , 3, {Masonry} )
  set_task_attributes(Painting , 2, {Ceiling} )
  set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

  ! Setting the resource consumptions
  consumes(Masonry , 7, money_available )

```

```
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[produces](#) [provides](#) [requires](#) [is_providing](#) [is_consuming](#) [is_requiring](#) [is_producing](#) [resusage](#)

requires

Purpose

Sets the minimal and maximal amount of resource required by a task for a particular resource. A task requires some amount of processing power of a resource when this amount must be made available for the execution of the task. The resource requirement is considered as renewable which means that whenever the task ends the processing power that was required is still available for processing other tasks. Note that a minimal amount of 0 is possible thus allowing modelisation of alternative resources. The type of resource requirement can be defined in several ways:

- By specifying a resource and a constant requirement parameter '*requirement*'. In this case the resource requirement is constant other the execution of the task.
- By specifying a resource and a minimal '*minReq*' and maximal '*maxReq*' requirement parameters: In this case the resource requirement is a decision variable but is constant other the execution of the task.
- By specifying a cpresusage '*usage*' (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*': This makes it possible to model alternative resources. The type of resource requirement is defined by the resource usage passed in argument (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*', a minimal '*minalt*' and a maximal '*maxalt*' number of alternative resources that are required: Between [*minalt*..*maxalt*] alternative requirements from the set will be active which means that several resources will be chosen as requirements by the task. The type of resource requirement is defined by the resource usage passed in argument (see [resusage](#) for further detail).

Synopsis

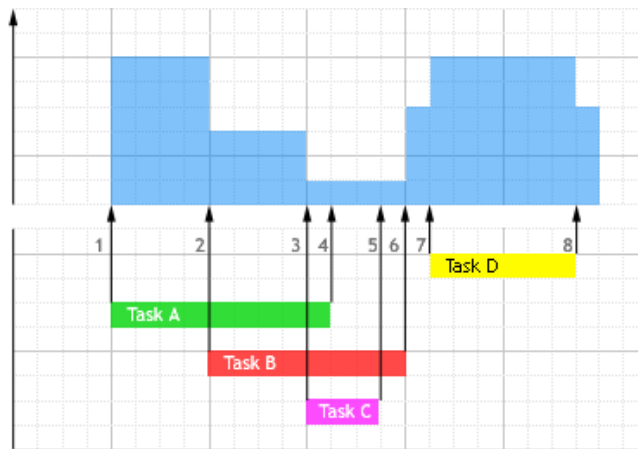
```
procedure requires(task:cptask, requirement:integer, resource:cpresource)
procedure requires(task:cptask, minReq:integer, maxReq:integer, resource:cpresource)
procedure requires(task:cptask, usage:cpresusage)
procedure requires(task:cptask, optUsages:set of cpresusage)
procedure requires(task:cptask, optUsages:set of cpresusage, minalt:integer,
                    maxalt:integer)
```

Arguments

task	the task
requirement	constant positive resource consumption
minReq	minimal resource requirement
maxReq	maximal resource requirement
resource	the resource to be required
usage	the resource usage
optUsages	the set of alternatives resource requirements
minalt	minimal number of resources that must be required
maxalt	maximal number of resources that may be required

Example

The following example illustrates this:



Task A produces 6 units of resource
 Task B requires 3 units of resource
 Task C consumes 2 units of resource
 Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```

model "Producer Consumer"
  uses "kalis"

  declarations
    Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
      Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
    money_available : cpresource                ! Resource declaration
  end-declarations

  ! 'money_available' is a cumulative resource with max. amount of 29$
  set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

  ! Limit resource availability to 20$ in the time interval [0,14]
  setcapacity(money_available, 0, 14, 20)

  ! Setting the task durations and predecessor sets
  set_task_attributes(Masonry , 7 )
  set_task_attributes(Carpentry, 3, {Masonry} )
  set_task_attributes(Roofing , 1, {Carpentry} )
  set_task_attributes(Windows , 1, {Roofing} )
  set_task_attributes(Facade , 2, {Roofing} )
  set_task_attributes(Garden , 1, {Roofing} )
  set_task_attributes(Plumbing , 8, {Masonry} )
  set_task_attributes(Ceiling , 3, {Masonry} )
  set_task_attributes(Painting , 2, {Ceiling} )
  set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

  ! Setting the resource consumptions
  consumes(Masonry , 7, money_available )

```

```
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[produces](#) [provides](#) [requires](#) [is_providing](#) [is_consuming](#) [is_requiring](#) [is_producing](#) [resusage](#)

produces

Purpose

Sets the minimal and maximal amount of resource produced by a task for a particular resource. A task produces some amount of processing power for a resource at its start. The capacity produced is considered as non renewable which means that the produced capacity remains available even after the completion of the task. Note that a minimal amount of 0 is possible thus allowing modelisation of alternative resources. The type of resource production can be defined in several ways:

- By specifying a resource and a constant production parameter '*production*'. In this case the resource production is constant other the execution of the task.
- By specifying a resource and a minimal '*minProd*' and maximal '*maxProd*' production parameters: In this case the resource production is a decision variable but is constant over the execution of the task.
- By specifying a cpresusage '*usage*' (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*': This makes it possible to model alternative resources. The type of resource production is defined by the resource usage passed in argument (see [resusage](#) for further detail).
- By a set of alternative cpresusage '*optUsages*', a minimal '*minalt*' and a maximal '*maxalt*' number of alternative resources that may be produced: Between [minalt..maxalt] alternative productions from the set will be active which means that several resources will be chosen to be produced by the task. The type of resource production is defined by the resource usage passed in argument (see [resusage](#) for further detail)

Synopsis

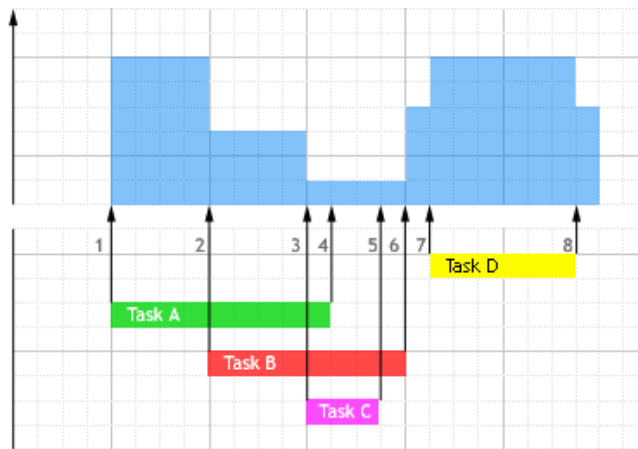
```
procedure produces(task:cptask, production:integer, resource:cpresource)
procedure produces(task:cptask,minProd:integer, maxProd:integer, resource:cpresource)
procedure produces(task:cptask, usage:cpresusage)
procedure produces(task:cptask, optUsages:set of cpresusage)
procedure produces(task:cptask, optUsages:set of cpresusage, minalt:integer,
                    maxalt:integer)
```

Arguments

task	the task
production	constant positive resource production
minProd	minimal resource production
maxProd	maximal resource production
resource	the resource to be produced
usage	the resource usage
optUsages	the set of alternatives resource production
minalt	minimal number of resources that must be produced
maxalt	maximal number of resources that may be produced

Example

The following example illustrates this:



Task A produces 6 units of resource

Task B requires 3 units of resource

Task C consumes 2 units of resource

Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```

model "Producer Consumer"
  uses "kalis"

  declarations
    Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
      Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
    money_available : cresource                 ! Resource declaration
  end-declarations

  ! 'money_available' is a cumulative resource with max. amount of 29$
  set_resource_attributes(money_available, KALIS_DISCRETE_RESOURCE, 29)

  ! Limit resource availability to 20$ in the time interval [0,14]
  set_capacity(money_available, 0, 14, 20)

  ! Setting the task durations and predecessor sets
  set_task_attributes(Masonry, 7)
  set_task_attributes(Carpentry, 3, {Masonry})
  set_task_attributes(Roofing, 1, {Carpentry})
  set_task_attributes(Windows, 1, {Roofing})
  set_task_attributes(Facade, 2, {Roofing})
  set_task_attributes(Garden, 1, {Roofing})
  set_task_attributes(Plumbing, 8, {Masonry})
  set_task_attributes(Ceiling, 3, {Masonry})
  set_task_attributes(Painting, 2, {Ceiling})
  set_task_attributes(MovingIn, 1, {Windows, Facade, Garden, Painting})

  ! Setting the resource consumptions
  consumes(Masonry, 7, money_available)

```

```
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[produces](#) [provides](#) [requires](#) [is_providing](#) [is_consuming](#) [is_requiring](#) [is_producing](#) [resusage](#)

provides

Purpose

Sets the minimal and maximal amount of resource provided by a task for a particular resource. A task provides some amount of processing power for a resource during its execution. The capacity provided is considered as renewable which means that whenever the providing task ends, the provided capacity is no longer available. Note that a minimal amount of 0 is possible thus allowing modeling of alternative resources. The type of resource provision can be defined in several ways:

- By specifying a resource and a constant provision parameter *'provision'*. In this case the resource provision is constant over the execution of the task.
- By specifying a resource and minimal *'minProv'* and maximal *'maxProv'* provision parameters: In this case the resource provision is a decision variable but is constant over the execution of the task.
- By specifying a cpresusage *'usage'* (see [resusage](#) for further detail).
- By a set of alternative cpresusage *'optUsages'*: This makes it possible to model alternative resources. The type of resource provision is defined by the resource usage passed in argument (see [resusage](#) for further detail).
- By a set of alternative cpresusage *'optUsages'*, a minimal *'minalt'* and a maximal *'maxalt'* number of alternative resources that must be provided: Between [minalt..maxalt] alternative provisions from the set will be active which means that several resources will be chosen to be provided by the task. The type of resource provision is defined by the resource usage passed in argument (see [resusage](#) for further detail).

Synopsis

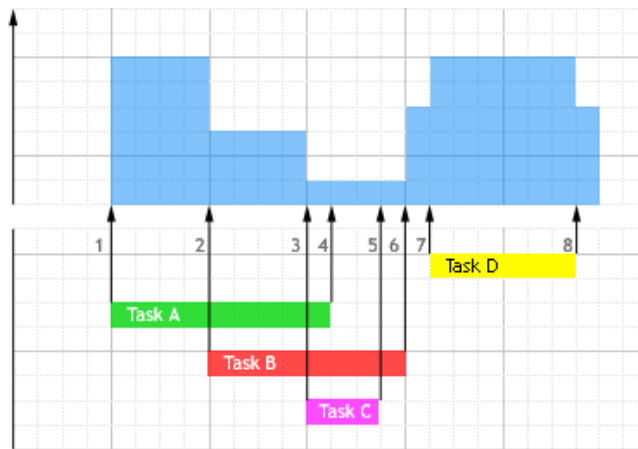
```
procedure provides(task:cptask, provision:integer, resource:cpresource)
procedure provides(task:cptask, minProv:integer, maxProv:integer, resource:cpresource)
procedure provides(task:cptask, usage:cpresusage)
procedure provides(task:cptask, optUsages:set of cpresusage)
procedure provides(task:cptask, optUsages:set of cpresusage, minalt:integer,
                    maxalt:integer)
```

Arguments

task	the task
provision	constant positive resource consumption
minProv	minimal resource consumption
maxProv	maximal resource provision
resource	the resource to be provided
usage	the resource usage
optUsages	the set of alternatives resource provision
minalt	minimal number of resources that must be provided
maxalt	maximal number of resources that may be provided

Example

The following example illustrates this:



Task A produces 6 units of resource
 Task B requires 3 units of resource
 Task C consumes 2 units of resource
 Task D provides 2 units of resource

- 1) Task A starts and produces 6 units of resource that will be available until the end of the schedule
- 2) Task B starts and requires 3 units of resource for its execution. The resource availability is now 3 units less.
- 3) Task C starts and consumes 2 units of resource. The resource availability is lowered by two units until the end of the schedule.
- 4) Task A ends and the resource availability remains constant as the production is definitive.
- 5) Task C ends and the resource availability remains constant as the consumption of resource is definitive.
- 6) Task B ends and 3 units of resource are available again since resource requirement is recoverable.
- 7) Task D starts and provides 2 units of resource during its execution.
- 8) Task D ends and the resource availability drop down of two units as the resource provision is recoverable.

```

model "Producer Consumer"
  uses "kalis"

  declarations
    Masonry, Carpentry, Roofing, Windows, Facade, Garden, Plumbing,
      Ceiling, Painting, MovingIn : cptask      ! Declaration of tasks
    money_available : cresource                 ! Resource declaration
  end-declarations

  ! 'money_available' is a cumulative resource with max. amount of 29$
  set_resource_attributes(money_available,KALIS_DISCRETE_RESOURCE,29)

  ! Limit resource availability to 20$ in the time interval [0,14]
  setcapacity(money_available, 0, 14, 20)

  ! Setting the task durations and predecessor sets
  set_task_attributes(Masonry , 7 )
  set_task_attributes(Carpentry, 3, {Masonry} )
  set_task_attributes(Roofing , 1, {Carpentry} )
  set_task_attributes(Windows , 1, {Roofing} )
  set_task_attributes(Facade , 2, {Roofing} )
  set_task_attributes(Garden , 1, {Roofing} )
  set_task_attributes(Plumbing , 8, {Masonry} )
  set_task_attributes(Ceiling , 3, {Masonry} )
  set_task_attributes(Painting , 2, {Ceiling} )
  set_task_attributes(MovingIn , 1, {Windows,Facade,Garden,Painting})

  ! Setting the resource consumptions
  consumes(Masonry , 7, money_available )

```

```
consumes(Carpentry, 3, money_available )
consumes(Roofing , 1, money_available )
consumes(Windows , 1, money_available )
consumes(Facade , 2, money_available )
consumes(Garden , 1, money_available )
consumes(Plumbing , 8, money_available )
consumes(Ceiling , 3, money_available )
consumes(Painting , 2, money_available )
consumes(MovingIn , 1, money_available )

! Find the optimal schedule (minimizing the makespan)
if cp_minimize(getmakespan) then
    cp_show_sol
else
    writeln("No solution found")
end-if

end-model
```

Related topics

[produces](#) [provides](#) [requires](#) [is_providing](#) [is_consuming](#) [is_requiring](#) [is_producing](#) [resusage](#)

set_resource_attributes

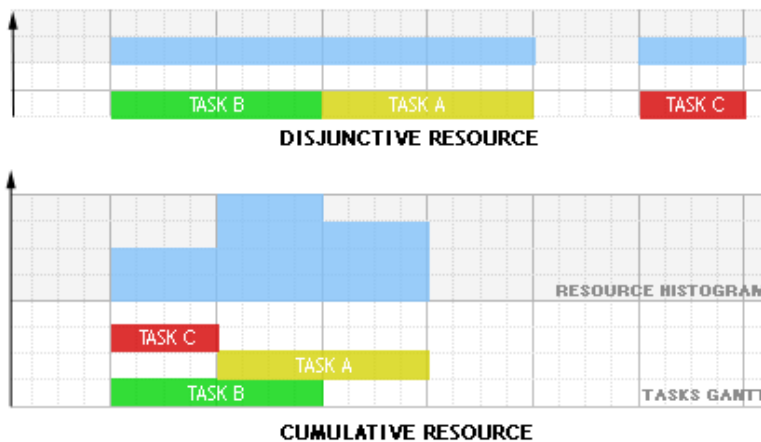
Purpose

Sets some attributes for a resource.

A resource can be of two different types:

- *Disjunctive* when the resource can be used only by one task at a time.
- *Cumulative* when the resource can be used by several tasks at the same time.

Note that a disjunctive resource is semantically equivalent to a cumulative resource with maximal capacity one and unit resource usage for each task using this resource but this equivalence does not hold in term of constraint propagation. The parameter type allows the user to choose between a disjunctive resource (KALIS_UNARY_RESOURCE) and a cumulative resource (KALIS_DISCRETE_RESOURCE). The last parameter, capacity, indicates the structural maximal capacity of the resource. When the resource is disjunctive this parameter must be equal to one. The structural capacity does not vary over time but a maximal temporal capacity can be imposed at any time point with the [setcapacity](#) function. The following graphic shows an example with three tasks A, B and C processed on a disjunctive resource and on a cumulative resource with resource usage 3 for task A, 1 for task B and 1 for task C.



Synopsis

```
procedure set_resource_attributes(resource:cpresource, type:integer)
procedure set_resource_attributes(resource:cpresource, type:integer, capacity:integer)
procedure set_resource_attributes(resource:cpresource, type:integer, capacity:integer,
    propagationHint:integer)
```

Arguments

resource	the resource
type	resource type: KALIS_DISCRETE_RESOURCE or KALIS_UNARY_RESOURCE
capacity	maximal capacity of a discrete resource
alg	propagation algorithm choice: KALIS_TIMETABLING , KALIS_TASK_INTERVALS , or KALIS_DISJUNCTIONS

Related topics

[set_task_attributes](#)

setcapacity

Purpose

Sets the maximal capacity (the maximal amount of processing units available) of a discrete resource between two time bounds. This routine can only be applied to resources of type KALIS_DISCRETE_RESOURCE.

Synopsis

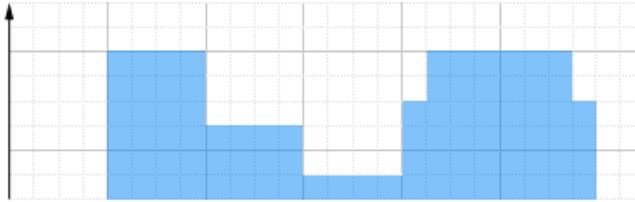
```
procedure setcapacity(resource:cpresource, tmin:integer, tmax:integer, capa:integer)
```

Arguments

resource	the resource
tmin	lower bound of the time interval
tmax	upper bound of the time interval
capa	the maximal amount of processing units available of resource in the interval [tmin,tmax]

Example

The following example shows how to use setcapacity:



```
model "Resource capacity"
  uses "kalis"

  declarations
    A, B, C : cptask          ! Declaration of tasks
    resource : cresource      ! Declaration of resource
  end-declarations

  setname(A,"A"); setname(B,"B"); setname(C,"C")

  ! The resource is a cumulative resource with maximum capacity 6
  set_resource_attributes(resource, KALIS_DISCRETE_RESOURCE, 6)

  ! Setting the resource capacity in the interval 0..2 to 3
  setcapacity(resource, 0, 2, 3)
  ! Setting the resource capacity in the interval 3..4 to 2
  setcapacity(resource, 3, 4, 2)
  ! Setting the resource capacity in time period 5 to 1
  setcapacity(resource, 5, 5, 1)

  ! Setting the task durations
  set_task_attributes(A, 1)
  set_task_attributes(B, 2)
  set_task_attributes(C, 3)

  ! Setting the resource requirements of the tasks
  requires(A, 1, resource)
  requires(B, 2, resource)
```

```
requires(C, 3, resource)

! Find the optimal schedule (minimizing the makespan)
if cp_schedule(getmakespan) <> 0 then
  cp_show_sol
else
  writeln("no solution found")
end-if

end-model
```

Related topics

[setmaxavailability](#) [setminusage](#) [set_resource_attributes](#)

setmaxavailability

Purpose

Sets the maximal capacity (the maximal amount of processing units available) of a resource between two time bounds. This routine can only be applied to resources of type KALIS_DISCRETE_RESOURCE.

Synopsis

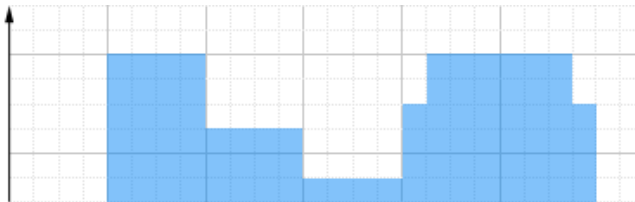
```
procedure setmaxavailability(resource:cpresource, tmin:integer, tmax:integer,
                             capa:integer)
```

Arguments

resource	the resource
tmin	lower bound of the time interval
tmax	upper bound of the time interval
capa	the maximal amount of processing units available of resource in the interval [tmin,tmax]

Example

The following example shows how to use setcapacity:



```
model "Resource capacity"
  uses "kalis"

  declarations
    A, B, C : cptask          ! Declaration of tasks
    resource : cresource      ! Declaration of resource
  end-declarations

  setname(A,"A"); setname(B,"B"); setname(C,"C")

  ! The resource is a cumulative resource with maximum capacity 6
  set_resource_attributes(resource, KALIS_DISCRETE_RESOURCE, 6)

  ! Setting the resource capacity in the interval 0..2 to 3
  setcapacity(resource, 0, 2, 3)
  ! Setting the resource capacity in the interval 3..4 to 2
  setcapacity(resource, 3, 4, 2)
  ! Setting the resource capacity in time period 5 to 1
  setcapacity(resource, 5, 5, 1)

  ! Setting the task durations
  set_task_attributes(A, 1)
  set_task_attributes(B, 2)
  set_task_attributes(C, 3)

  ! Setting the resource requirements of the tasks
  requires(A, 1, resource)
```

```
requires(B, 2, resource)
requires(C, 3, resource)

! Find the optimal schedule (minimizing the makespan)
if cp_schedule(getmakespan) <> 0 then
  cp_show_sol
else
  writeln("no solution found")
end-if

end-model
```

Related topics

[setcapacity](#) [setminusage](#) [set_resource_attributes](#)

setminusage

Purpose

Sets the minimum usage (the minimum number of processing units used) of a resource between two time bounds. This routine can only be applied to resources of type `KALIS_DISCRETE_RESOURCE`.

Synopsis

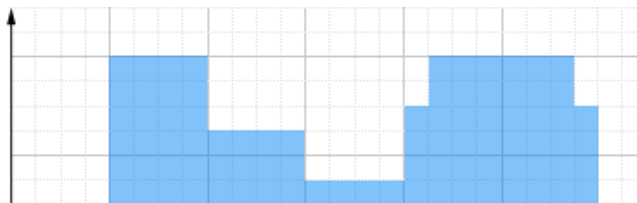
```
procedure setminusage(resource:cpresource, tmin:integer, tmax:integer, capa:integer)
```

Arguments

<code>resource</code>	the resource
<code>tmin</code>	lower bound of the time interval
<code>tmax</code>	upper bound of the time interval
<code>capa</code>	the minimum usage of processing units of resource in the interval [tmin,tmax]

Example

The following example shows how to use `setcapacity`:



```
model "Resource capacity"
  uses "kalis"

  declarations
    A, B, C : cptask          ! Declaration of tasks
    resource : cresource      ! Declaration of resource
  end-declarations

  setname(A,"A"); setname(B,"B"); setname(C,"C")

  ! The resource is a cumulative resource with maximum capacity 6
  set_resource_attributes(resource, KALIS_DISCRETE_RESOURCE, 6)

  ! Setting the resource capacity in the interval 0..2 to 3
  setcapacity(resource, 0, 2, 3)
  ! Setting the resource capacity in the interval 3..4 to 2
  setcapacity(resource, 3, 4, 2)
  ! Setting the resource capacity in time period 5 to 1
  setcapacity(resource, 5, 5, 1)

  ! Setting the task durations
  set_task_attributes(A, 1)
  set_task_attributes(B, 2)
  set_task_attributes(C, 3)

  ! Setting the resource requirements of the tasks
  requires(A, 1, resource)
  requires(B, 2, resource)
  requires(C, 3, resource)
```

```
! Find the optimal schedule (minimizing the makespan)
if cp_schedule(getmakespan) <> 0 then
  cp_show_sol
else
  writeln("no solution found")
end-if

end-model
```

Related topics

[setcapacity](#) [setmaxavailability](#) [set_resource_attributes](#)

getcapacity

Purpose

Gets the maximal capacity (the maximal amount of processing units available) of a resource between two time bounds.

Synopsis

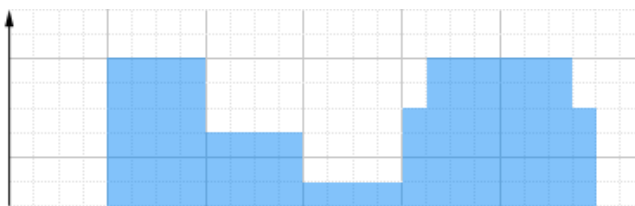
```
function getcapacity(resource:cpresource, tmin:integer, tmax:integer, capa:integer) :
    integer
```

Arguments

resource the resource
tslot lowerbound of the time interval

Example

The following example shows how to use getcapacity:



```
model "Resource capacity"
    uses "kalis"

    declarations
        A, B, C : cptask          ! Declaration of tasks
        resource : cprsource       ! Declaration of resource
    end-declarations

    setname(A,"A"); setname(B,"B"); setname(C,"C")

    ! The resource is a cumulative resource with maximum capacity 6
    set_resource_attributes(resource, KALIS_DISCRETE_RESOURCE, 6)

    ! Setting the resource capacity in the interval 0..2 to 3
    setcapacity(resource, 0, 2, 3)
    ! Setting the resource capacity in the interval 3..4 to 2
    setcapacity(resource, 3, 4, 2)
    ! Setting the resource capacity in time period 5 to 1
    setcapacity(resource, 5, 5, 1)

    ! Setting the task durations
    set_task_attributes(A, 1)
    set_task_attributes(B, 2)
    set_task_attributes(C, 3)

    ! Setting the resource requirements of the tasks
    requires(A, 1, resource)
    requires(B, 2, resource)
    requires(C, 3, resource)

    ! Find the optimal schedule (minimizing the makespan)
    if cp_schedule(getmakespan) <> 0 then
```

```
        cp_show_sol  
    else  
        writeln("no solution found")  
    end-if  
  
end-model
```

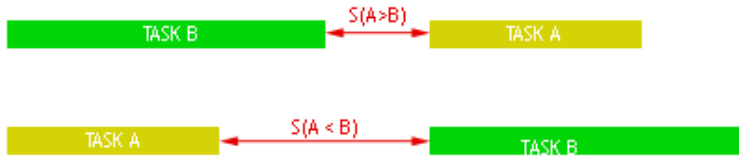
Related topics

[setcapacity](#)

setuptime

Purpose

Sets sequence dependent setup times between two tasks. The setup time depends on the relative execution order in the schedule of the two tasks.



Synopsis

```
procedure setuptime(task1:cptask, task2:cptask, task1_before_task2:integer,  
                    task2_before_task1:integer)
```

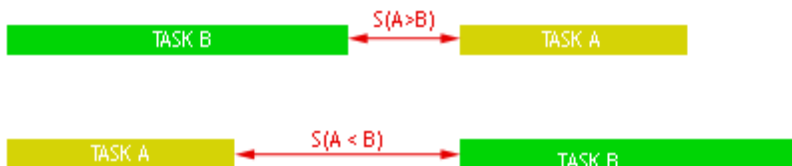
Arguments

task1	a task
task2	another task
task1_before_task2	setup time between the two tasks if task1 preceedes task2
task2_before_task1	setup time between the two tasks if task2 preceedes task1

getsetuptime

Purpose

Return the sequence dependent setup times between task1 and task2 if task1 precedes task2. The setup time depends on the relative execution order in the schedule of the two tasks.



Synopsis

```
function getsetuptime(task1:cptask, task2:cptask) : integer
```

Arguments

task1 a task
task2 another task

is_fixed

Purpose

Returns `true` if the status of the disjunction passed in argument is known.

Synopsis

```
function is_fixed(disj:cpctr) : boolean
```

Argument

`disj` the disjunction

Return value

`true` if disjunction is fixed

Example

The following example shows how to see if a cpctr `disj` is fixed

```
if is_fixed(disj) then
  write('status of disj is known')
end-if
```

Related topics

[setpriority](#) [getpriority](#)

cp_set_schedule_strategy

Purpose

The `cp_schedule` search algorithm is decomposed in two distinct phases. The first phase target the finding of an near optimal heuristic solution quickly. The second and last phase focuses on improving the bound and prove optimality.

Synopsis

```
procedure cp_set_schedule_strategy(phase:integer, arrayBranching:array(range) of
    cpbranching)
procedure cp_set_schedule_strategy(phase:integer, branching:cpbranching)
```

Arguments

phase	algorithm phase <code>KALIS_INITIAL_SOLUTION</code> , <code>KALIS_OPTIMAL_SOLUTION</code>
branching	the search strategy
arrayBranching	the search strategies

Example

The following example shows how to set the search stragey for the `cp_schedule` function

```
cp_set_schedule_strategy(KALIS_INITIAL_SOLUTION,
    assign_var(KALIS_SMALLEST_MIN,KALIS_MIN_TO_MAX))
```

Related topics

`KALIS_INITIAL_SOLUTION` `KALIS_OPTIMAL_SOLUTION` `cp_schedule`

cp_get_default_schedule_strategy

Purpose

The `cp_schedule` search algorithm is decomposed in two distinct phases. The target of the first phase is finding quickly a near optimal heuristic solution. The second and last phase focuses on improving the bound and proving optimality.

Synopsis

```
procedure cp_get_default_schedule_strategy(phase:integer, branching:cpbranching)
```

Arguments

phase	algorithm phase (<code>KALIS_INITIAL_SOLUTION</code> , <code>KALIS_OPTIMAL_SOLUTION</code>)
branching	the search strategy

Example

The following example shows how to set the search strategy for the `cp_schedule` function

```
cp_set_schedule_strategy(KALIS_INITIAL_SOLUTION,  
                        assign_var(KALIS_SMALLEST_MIN,KALIS_MIN_TO_MAX))
```

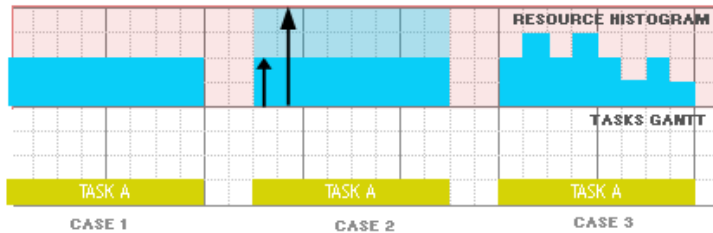
Related topics

`KALIS_INITIAL_SOLUTION` `KALIS_OPTIMAL_SOLUTION` `cp_schedule`

resusage

Purpose

Creates a resource usage. A resource usage is a means to specify the way that a resource will be produced/consumed/provided/required by a task. Three types of resource usages can be used:



- **CASE 1: Constant resource usage.** This is the simplest case where the resource usage is constant all along the execution of the task. For example a task A will require 2 units of resource R during its execution.
- **CASE 2: Variable resource usage but constant over the execution of the task.** This case allows the resource usage to be variable but constant during the execution of the task. Therefore, a decision variable is associated with the resource usage that can be constrained like any other variable. For example a task A will require between 2 units and 4 units of resource R during its execution.
- **CASE 3: Variable resource usage over the execution of the task.** This case allows the resource usage to vary during the execution of the task using a resource usage profile defined as a list of integers. For example a task A will require [2,3,2,3,2,1,2,1] resource units which means '2' for the first timestep of its execution; '3' for the second etc.. If the duration of the task exceeds the length of the profile, then the resource usage is considered as 0 after the end of the profile. If the task duration is smaller than the length of the profile, the profile will be truncated to the duration of the task.

Synopsis

```
function resusage(resource:cpresource, usage:integer) : cpresusage
function resusage(resource:cpresource, usagemin:integer, usagemax:integer) :
    cpresusage
function resusage(resource:cpresource, profile:array of integer) : cpresusage
```

Arguments

```
resource    the resource
usage       the constant resource usage
usagemin    minimal resource usage
usagemax    maximal resource usage
profile     profile of resource usage
```

Example

The following example illustrates this:

```
model "Non constant resource usage"
uses "kalis"

forward procedure print_solution(tasks: set of cptask,
                                resources: set of cpresource)

declarations
    res1,res2: cpresource
```

```

    taska,taskb,taskc: cptask
end-declarations

taska.start    <= 15
taska.duration = 4
taskb.start    <= 15
taskb.duration = 3
taskc.start    <= 15
taskc.duration = 5

! Define a discrete resource with periods of unavailability
set_resource_attributes(res1, KALIS_DISCRETE_RESOURCE, 6)
setcapacity(res1, 0, 15, 0)
setcapacity(res1, 1, 5, 6)
setcapacity(res1, 7, 11, 6)

requires(taska, {resusage(res1,[1,3,5,6])})
requires(taskb, {resusage(res1,[5,3,1])})
requires(taskc, {resusage(res1,1,1)})

! Define a resource with initial capacity at 0
set_resource_attributes(res2, KALIS_DISCRETE_RESOURCE, 0)

provides(taska, resusage(res2,[1,3,1,2]))
consumes(taskb, resusage(res2,[3,1,2]))
produces(taskc, resusage(res2,[3,1,2,0,2]))

if not cp_propagate then
    writeln("Problem is infeasible"); exit(1)
end-if

while (cp_find_next_sol) do
    print_solution({taska,taskb,taskc},{res1,res2})
end-do

!-----
!**** Display results ****
procedure print_solution(tasks: set of cptask,
                        resources: set of cpresource)

forall(res in resources) do
    writeln("Resource ", res.name)
    forall(timeindex in 0..getub(getmakespan)) do
        write(strfmt(timeindex,3), " Cap: ", getcapacity(res,timeindex))

        forall(t in tasks)
            if getrequirement(t,res,timeindex)>0 then
                write(" ", t.name, "(req):", getrequirement(t,res,timeindex))
            elif getproduction(t,res,timeindex)>0 then
                write(" ", t.name, "(prod):", getproduction(t,res,timeindex))
            elif getconsumption(t,res,timeindex)>0 then
                write(" ", t.name, "(cons):", getconsumption(t,res,timeindex))
            elif getprovision(t,res,timeindex)>0 then
                write(" ", t.name, "(prov):", getprovision(t,res,timeindex))
            end-if
        end-if
    end-forall
end-forall

```

```
writeln  
end-do  
end-do  
  
end-procedure  
  
end-model
```

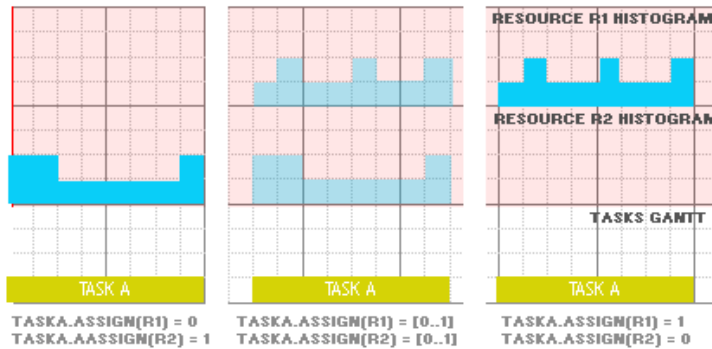
Related topics

[set_task_attributes](#)

getassignment

Purpose

Gets the cpvar representing the assignment of a task for a particular resource. In the following example, either task A requires resource R1 with resource profile [1,2,1,1,2,1,1,2], or task A requires resource R2 with resource profile [2,2,1,1,1,1,1,2]. The choice depends on the value of the decision variables A.assignment(R1) and A.assignment(R2) that can both take a value of 0 (the task is not assigned to this resource) or a value of 1 (the task is assigned to this resource).



Synopsis

```
function getassignment(task:cptask, resource:cpresource) : cpvar
```

Arguments

task the task
resource the resource

Return value

The cpvar representing the assignment of 'task' to 'resource'

Example

The following example illustrates this:

```
model "Alternative resources and non constant resource usage"
uses "kalis"

declarations
    res1,res2    : cresource
    taska,taskb  : cptask
    arr1,arr2    : list of integer
end-declarations

! Fix start times and durations
taska.start     = 3
taska.duration  = 4
taskb.start     = 3
taskb.duration  = 4

! Define 2 cumulative resources
set_resource_attributes(res1, KALIS_DISCRETE_RESOURCE, 4)
set_resource_attributes(res2, KALIS_DISCRETE_RESOURCE, 4)

setname(taska,"taska"); setname(taskb,"taskb")
setname(res1,"R1"); setname(res2,"R2")
```

```
! Define alternative resources for both tasks
arr1 := [1,3,2,3]
arr2 := [2,4,1,3]
requires(taska, {resusage(res1,arr1),resusage(res2,arr2)}, 1, 1)
requires(taskb, {resusage(res1,1,1),resusage(res2,1,1)}, 1, 1)

! Find all solutions
while (cp_find_next_sol) do
  cp_show_sol
end-do

end-model
```

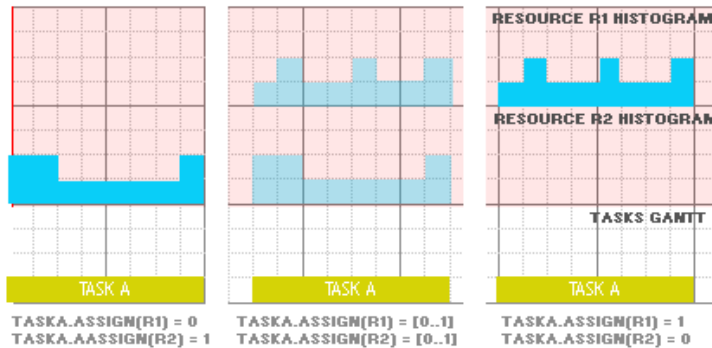
Related topics

[has_assignment](#) [getstart](#) [getend](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getproduction](#)
[getprovision](#)

has_assignment

Purpose

In the following example, either task A requires resource R1 with resource profile [1,2,1,1,2,1,1,2], or task A requires resource R2 with resource profile [2,2,1,1,1,1,1,2]. The choice depends on the value of the decision variables A.assignment(R1) and A.assignment(R2) that can both take a value of 0 (the task is not assigned to this resource) or a value of 1 (the task is assigned to this resource).



Synopsis

```
function has_assignment(task:cptask, resource:cpresource) : boolean
```

Arguments

task the task
resource the resource

Return value

true IFF a cpvar representing the assignment of 'task' to 'resource' exists

Example

The following example illustrates this:

```
model "Alternative resources and non constant resource usage"
uses "kalis"

declarations
  res1,res2   : cresource
  taska,taskb : cptask
  arr1,arr2   : list of integer
end-declarations

! Fix start times and durations
taska.start   = 3
taska.duration = 4
taskb.start   = 3
taskb.duration = 4

! Define 2 cumulative resources
set_resource_attributes(res1, KALIS_DISCRETE_RESOURCE, 4)
set_resource_attributes(res2, KALIS_DISCRETE_RESOURCE, 4)

setname(taska,"taska"); setname(taskb,"taskb")
setname(res1,"R1"); setname(res2,"R2")

! Define alternative resources for both tasks
```

```
arr1 := [1,3,2,3]
arr2 := [2,4,1,3]
requires(taska, {resusage(res1,arr1),resusage(res2,arr2)}, 1, 1)
requires(taskb, {resusage(res1,1,1),resusage(res2,1,1)}, 1, 1)

! Find all solutions
while (cp_find_next_sol) do
  cp_show_sol
end-do

end-model
```

Related topics

[getassignment](#) [getstart](#) [getend](#) [getduration](#) [getconsumption](#) [getrequirement](#) [getproduction](#)
[getprovision](#)

setidletimes

Purpose

Specifies the set of timesteps during which a resource should be considered as idle. When a task overlaps an idle timestep, resource usages are shifted to the right provided that the task duration is variable.

Synopsis

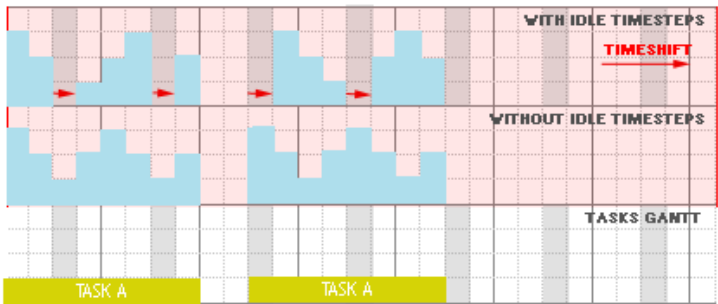
```
procedure setidletimes(resource:cpresource, idleTimeSteps: set of integer)
```

Arguments

- resource the resource
- idleTimeSteps The set of idle time steps

Example

Here is an example:



Related topics

[consumes](#) [requires](#) [provides](#) [produces](#) [resusage](#) [is_idletime](#)

is_idletime

Purpose

Tests whether a specific timestep is defined as an idle time for the resource passed in the argument.

Synopsis

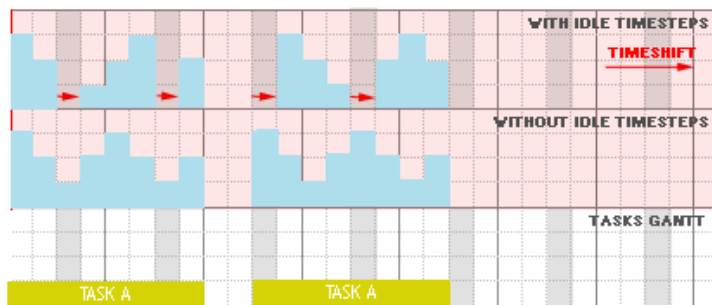
```
function is_idletime(resource:cpresource, timestep: integer) : boolean
```

Arguments

resource the resource
timestep The time step

Example

Here is an example:



Related topics

[consumes](#) [requires](#) [provides](#) [produces](#) [resusage](#) [is_idletime](#)

cp_close_schedule

Purpose

Close the schedule, *i.e.*, mark problem definition as complete, and post the underlying structural constraints of the tasks and resources. (A call to `propagate` does not automatically close the schedule.)

Synopsis

```
procedure cp_close_schedule
```

9.8 Linear relaxations

<code>cp_add_linrelax_solver</code>	Add a linear relaxation solver to the linear relaxation solver list	p. 277
<code>cp_clear_linrelax_solver</code>	Clear the linear relaxation solver list	p. 279
<code>cp_get_linrelax</code>	Returns an automatic relaxation of the cp problem	p. 272
<code>cp_remove_linrelax_solver</code>	Remove a linear relaxation solver from the linear relaxation solver list	p. 278
<code>cp_show_relax</code>	Pretty printing of a linear relaxation	p. 276
<code>export_prob</code>	Export the linear relaxation in LP format	p. 285
<code>fix_to_relaxed</code>	Fix the continuous variables to their optimal value in the relaxation solver passed in argument	p. 274
<code>generate_cuts</code>	Generate and add cuts to the relaxation passed in parameters	p. 280
<code>get_indicator</code>	Get an indicator variable for a given variable and a value.	p. 283
<code>get_linrelax</code>	Get the linear relaxation for a constraint	p. 284
<code>get_linrelax_solver</code>	Returns a linear relaxation solver from a linear relaxation, an objective variables and some configuration parameters	p. 273
<code>get_reduced_cost</code>	Get a reduced cost value from a linear relaxation solver	p. 286
<code>get_relaxed_value</code>	Returns the optimal relaxed value for a variable in a relaxation	p. 287
<code>KALIS_LARGEST_REDUCED_COST</code>	Get a largest reduced cost variable selector from a linear relaxation solver	p. 270
<code>KALIS_NEAREST_RELAXED_VALUE</code>	Get a nearest relaxed value selector from a linear relaxation solver	p. 271
<code>lp_optimize</code>	Launch LP/MIP solver without CP branching.	p. 281
<code>set_integer</code>	Set integrality flag for a variable in a linear relaxation	p. 288
<code>set_linrelax_solver_attribute</code>	Parameter setting for a linear relaxation solver.	p. 282
<code>set_verbose_level</code>	Set the verbose level for a specific linear relaxation solver	p. 275

KALIS_LARGEST_REDUCED_COST

Purpose

Get a largest reduced cost variable selector from a linear relaxation solver.

Synopsis

```
procedure KALIS_LARGEST_REDUCED_COST(relaxationSolver:cpInrelaxsolver)
```

Argument

`relaxationSolver` the linear relaxation solver

Example

The following example shows how to use this function with a specific relaxation solver

```
cp_set_branching(assign_var(KALIS_LARGEST_REDUCED_COST(mysolver),  
                           KALIS_NEAREST_RELAXED_VALUE(mysolver)))
```

Related topics

[KALIS_NEAREST_RELAXED_VALUE](#) [cp_set_branching](#)

KALIS_NEAREST_RELAXED_VALUE

Purpose

Get a nearest relaxed value selector from a linear relaxation solver.

Synopsis

```
procedure KALIS_NEAREST_RELAXED_VALUE(relaxationSolver:cplinrelaxsolver)
```

Argument

`relaxationSolver` the linear relaxation solver

Example

The following example shows how to use this function with a specific `cplinrelaxsolver`

```
cp_set_branching(assign_var(KALIS_LARGEST_REDUCED_COST(mysolver),  
                             KALIS_NEAREST_RELAXED_VALUE(mysolver)))
```

Related topics

[KALIS_LARGEST_REDUCED_COST](#) [cp_set_branching](#)

cp_get_linrelax

Purpose

Returns an automatic relaxation of the cp problem

Synopsis

```
function cp_get_linrelax(orientation: integer) : cplinrelax
function cp_get_linrelax(orientation: integer,constraints: set of integer) :
    cplinrelax
```

Arguments

orientation	0 for an 'LP oriented' relaxation (convex hull) and 1 for a 'MIP oriented' relaxation (can be an exact representation of the underlying CP problem)
constraints	the set of constraints types to relax (KALIS_LINEAR_CONSTRAINTS , KALIS_LOGICAL_CONSTRAINTS , KALIS_DISTANCE_CONSTRAINTS , KALIS_NON_LINEAR_CONSTRAINTS , KALIS_ALL_CONSTRAINTS)

Return value

A linear relaxation

Example

The following example shows how to obtain a 'MIP oriented' relaxation by relaxing only the linear part of the problem and the logical constraints.

```
relax := cp_get_linrelax(1,{KALIS_LINEAR_CONSTRAINTS,
                           KALIS_LOGICAL_CONSTRAINTS})
```

Related topics

[get_linrelax_solver](#) [cp_add_linrelax_solver](#)

get_linrelax_solver

Purpose

Returns a linear relaxation solver from a linear relaxation, an objective variables and some configuration parameters

Synopsis

```
function get_linrelax_solver(linrelax:cplinrelax, objective:cpvar, sense:integer,
    solvingType:integer, before_event:proc, must_relax:proc, after_event:proc) :
    cplinrelaxsolver
function get_linrelax_solver(linrelax:cplinrelax, objective:cpfloatvar, sense:integer,
    solvingType:integer, before_event:proc, must_relax:proc, after_event:proc) :
    cplinrelaxsolver
function get_linrelax_solver(linrelax:cplinrelax, objective:cpvar, sense:integer,
    solvingType:integer, configuration:integer) : cplinrelaxsolver
function get_linrelax_solver(linrelax:cplinrelax, objective:cpfloatvar, sense:integer,
    solvingType:integer, configuration:integer) : cplinrelaxsolver
function get_linrelax_solver(linrelax:cplinrelax, objective:cpvar, sense:integer,
    solvingType:integer, cpvarsToBeInstantiated:set of cpvar,
    floatvarsToBeInstantiated:set of cpfloatvar) : cplinrelaxsolver
function get_linrelax_solver(linrelax:cplinrelax, objective:cpfloatvar, sense:integer,
    solvingType:integer, cpvarsToBeInstantiated:set of cpvar,
    floatvarsToBeInstantiated:set of cpfloatvar) : cplinrelaxsolver
```

Arguments

relax	the linear relaxation
objective	the objective variable
sense	KALIS_MINIMIZE for minimization or KALIS_MAXIMIZE for maximization
solvingType	KALIS_SOLVE_AS_LP or KALIS_SOLVE_AS_MIP
before_event	a user callback triggered before the relaxation is solved
must_relax	a user callback saying when to solve the relaxation
after_relax	a user callback triggered after the resolution of the relaxation
configuration	One of the three predefined configurations (KALIS_TOPNODE_RELAX_SOLVER , KALIS_TREENODE_RELAX_SOLVER , or KALIS_BILEVEL_RELAX_SOLVER)
cpvarsToBeInstantiated	the set of cpvar to be instantiated before the resolution of the relaxation for a KALIS_BILEVEL_RELAX_SOLVER
floatvarsToBeInstantiated	the set of cpfloatvar to be instantiated before the resolution of the relaxation for a KALIS_BILEVEL_RELAX_SOLVER

Return value

A linear relaxation solver

fix_to_relaxed

Purpose

This method can be called during the tree search process to instantiate all the continuous variables to their values in the optimal solution of a relaxation.

Synopsis

```
procedure fix_to_relaxed(linrelaxSolver:cpllinrelax)
```

Argument

linrelaxSolver the linear relaxation solver

Example

The following example shows how to use the fix_to_relaxed method.

```
fix_to_relaxed(linrelaxSolver)
```

Related topics

[get_relaxed_value](#)

set_verbose_level

Purpose

Set the verbose level for a specific linear relaxation solver.

Synopsis

```
procedure set_verbose_level(linrelaxSolver:cplinrelaxsolver,flag:boolean)
```

Arguments

linrelaxSolver	the linear relaxation solver
flag	value true or false to enable/disable output printing

Example

The following example shows how to use the set_verbose_level function.

```
set_verbose_level(linrelaxSolver,true)
```

Related topics

[get_linrelax_solver](#)

cp_show_relax

Purpose

Pretty printing of the linear relaxation to the standard output.

Synopsis

```
procedure cp_show_relax(relax:cplinrelax)
```

Argument

`relax` the linear relaxation to be printed

Example

The following example shows how to use the `cp_show_relax` method.

```
cp_show_relax(relax)
```

Related topics

[cp_show_var](#) [cp_show_stats](#) [cp_show_prob](#)

cp_add_linrelax_solver

Purpose

Add a linear relaxation solver to the linear relaxation solver list.

Synopsis

```
procedure cp_add_linrelax_solver(linrelaxSolver:cplinrelaxsolver)
```

Argument

linrelaxSolver the linear relaxation solver to add

Example

The following example shows how to use the cp_add_linrelax_solver function.

```
cp_add_linrelax_solver(mylinrelaxsolver)
```

Related topics

[get_linrelax_solver](#) [cp_clear_linrelax_solver](#) [cp_remove_linrelax_solver](#)

cp_remove_linrelax_solver

Purpose

Remove a linear relaxation solver from the linear relaxation solver list.

Synopsis

```
procedure cp_remove_linrelax_solver(linrelaxSolver:cplinrelaxsolver)
```

Argument

linrelaxSolver the linear relaxation solver to remove

Example

The following example shows how to use the cp_remove_linrelax_solver function.

```
cp_remove_linrelax_solver(linrelaxsolver)
```

Related topics

[get_linrelax_solver](#) [cp_clear_linrelax_solver](#) [cp_add_linrelax_solver](#)

cp_clear_linrelax_solver

Purpose

Clear the linear relaxation solver list.

Synopsis

```
procedure cp_clear_linrelax_solver
```

Example

The following example shows how to use the cp_remove_linrelax_solver function.

```
cp_clear_linrelax_solver
```

Related topics

[get_linrelax_solver](#) [cp_remove_linrelax_solver](#) [cp_add_linrelax_solver](#)

generate_cuts

Purpose

Generate and add cuts to the relaxation passed in parameters

Synopsis

```
procedure generate_cuts(linrelaxsolver:cplinrelaxsolver, linrelax:cplinrelax)
```

Arguments

linrelaxsolver	a linear relaxation solver
linrelax	the linear relaxation to use

Related topics

[get_linrelax_solver](#) [cp_get_linrelax](#) [get_linrelax](#)

lp_optimize

Purpose

Launch optimization of the relaxation solver with the LP/MIP solver alone (no branching by Kalis).

Synopsis

```
function lp_optimize(linrelaxsolver:cplinrelaxsolver, sense: integer, alg: integer)
    : integer
```

Arguments

linrelaxsolver	a linear relaxation solver
sense	KALIS_MINIMIZE for minimization or KALIS_MAXIMIZE for maximization
alg	solve as LP (0) or as MIP (1)

Related topics

[get_linrelax_solver](#) [cp_get_linrelax](#) [get_linrelax](#)

set_linrelax_solver_attribute

Purpose

Sets the value of the indicated control parameter of a linear relaxation solver.

Synopsis

```
procedure set_linrelax_solver_attribute(linrelaxSolver:cpllinrelaxsolver,  
    param:integer, value:integer)  
procedure set_linrelax_solver_attribute(linrelaxSolver:cpllinrelaxsolver,  
    param:integer, value:integer)
```

Arguments

linrelaxSolver	a linear relaxation solver
param	identifier of a solver parameter (<code>KALIS_RELAX_PRESOLVE</code> , <code>KALIS_RELAX_RCOSTS_PROPAG</code> , <code>KALIS_RELAX_RELOAD_BASIS</code> , <code>KALIS_RELAX_GLOBAL</code> , <code>KALIS_RELAX_ALGORITHM</code>)
value	the value to be set (0 or 1 in most cases, values for <code>KALIS_RELAX_ALGORITHM</code> : <code>KALIS_PRIMAL_SIMPLEX</code> , <code>KALIS_DUAL_SIMPLEX</code> , <code>KALIS_BARRIER</code> , <code>KALIS_NETWORK_SIMPLEX</code>)

Example

The following example shows how to set the solution algorithm for a linear relaxation solver.

```
set_linrelax_solver_attribute(mysolver, KALIS_RELAX_ALGORITHM,  
    KALIS_PRIMAL_SIMPLEX)
```

Related topics

`get_linrelax_solver` `cp_remove_linrelax_solver` `cp_add_linrelax_solver`

get_indicator

Purpose

Get a 0-1 valued cpauxvar indicating whether the variable is instantiated (1) or not (0) to the specified value.

Synopsis

```
function get_indicator(var:cpvar,i:integer) :  cpauxvar
```

Arguments

var	the variable
i	the value

Return value

An auxiliary relaxation variable

Example

The following example shows how to use the get_indicator function.

```
indvar:= get_indicator(var,i)
```

get_linrelax

Purpose

Get the linear relaxation for a constraint.

Synopsis

```
function get_linrelax(ctr:cpctr, orientation:integer) : cplinrelax
```

Arguments

ctr	the constraint to relax
orientation	0 for a 'LP oriented' relaxation and 1 for a 'MIP oriented' one

Return value

A linear relaxation

Example

The following example shows how to get a 'MIP oriented' linear relaxation of a constraint 'ctr'.

```
get_linrelax(ctr,1)
```

Related topics

[cp_get_linrelax](#)

export_prob

Purpose

Export the linear program from a linear relaxation solver to a text file in LP format.

Synopsis

```
procedure export_prob(linrelaxsolver:cplinrelaxsolver, name:string)
```

Arguments

linrelaxsolver	the linear relaxation solver
name	a file name

Example

The following example shows how to use the export_prob function.

```
export_prob(mylinrelaxsolver, "mymatout")
```

Related topics

[cp_get_linrelax](#) [get_linrelax_solver](#) [get_linrelax](#)

get_reduced_cost

Purpose

Get the reduced cost for a variable from a linear relaxation solver

Synopsis

```
function get_reduced_cost(linsolver:cpllinrelaxsolver, var:cpvar) : real
function get_reduced_cost(linsolver:cpllinrelaxsolver, var:cpfloatvar) : real
```

Arguments

<code>linsolver</code>	the linear relaxation solver
<code>var</code>	a CP variable

Return value

The reduced cost value

Example

The following example shows how to use the `get_reduced_cost` function.

```
rc := get_reduced_cost(linsolver, x)
```

Related topics

[get_relaxed_value](#) [get_linrelax](#) [set_integer](#)

get_relaxed_value

Purpose

Returns the optimal relaxed value for a variable in a relaxation.

Synopsis

```
function get_relaxed_value(linsolver:cpllinrelaxsolver, var:cpvar) : real
function get_relaxed_value(linsolver:cpllinrelaxsolver, var:cpfloatvar) : real
```

Arguments

<code>linsolver</code>	the linear relaxation solver
<code>var</code>	a CP variable

Return value

The relaxation solution value

Example

The following example shows how to use the `get_relaxed_value` function.

```
rv := get_relaxed_value(linsolver, x)
```

Related topics

[get_reduced_cost](#) [get_linrelax](#) [set_integer](#)

set_integer

Purpose

This method turns a variable into an integer variable in a linear relaxation.

Synopsis

```
procedure set_integer(relax:cpllinrelax, var:cpauxvar, global:boolean)
procedure set_integer(relax:cpllinrelax, var:cpvar, global:boolean)
```

Arguments

relax	a linear relaxation
var	a relaxation variable
global	true for a discrete variable, false otherwise

Example

The following example shows how to use the set_integer routine.

```
set_integer(myrelax, xaux, true)
```

Related topics

[get_relaxed_value](#) [get_reduced_cost](#) [get_linrelax](#)

APPENDIX A

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

Please include 'Xpress' in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718

Fax: +44 870 420 3601

Xpress Optimization, FICO

FICO House

International Square

Starley Way

Birmingham B37 7GN

UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count™.

Bibliography

- [BR01] C. Bessière and J.-C. Régin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of IJCAI-01*, pages 309—315, Seattle, 2001.
- [CL94a] Y. Caseau and F. Laburthe. Cumulative Scheduling with Task Intervals. In *Joint International Conference and Symposium on Logic Programming*, 1994.
- [CL94b] Y. Caseau and F. Laburthe. Improved CLP Scheduling with Task Intervals. In *ICLP94 Proceedings of the Eleventh International Conference on Logic Programming, MIT Press*, pages 369—383, 1994.
- [LP96] C. Le Pape. Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative. In *Proceedings of the 15th Workshop of the U.K. Planning and Scheduling Special Interest Group*, pages 17—22, Liverpool, 1996.
- [MVH08] L. Mercier and P. Van Hentenryck. Edge Finding for Cumulative Scheduling. *INFORMS Journal on Computing*, 20:143—153, 2008.
- [R94] J.-C. Régin. A Filtering Algorithm for Constraint of Difference in CSPs. In *Proceedings of AAAI94*, pages 362—367, Seattle, 1994.

Index

A

- abs, 81
- addpredecessors, 211
- addsuccessors, 213
- all_different, 83
- and, 69
- arity, 1
- assign_and_forbid, 172
- assign_var, 168
- automatic relaxation, 21

B

- benders decomposition, 24
- branching
 - relaxation, 24
- branching scheme, 9
- bs_group, 193

C

- constraint graph, 1
- Constraint Programming, 1
- constraint propagation, 2
- constraint satisfaction problem, 1
- consumes, 233
- contains, 128
- CP, see Constraint Programming
- cp_add_linrelax_solver, 277
- cp_clear_linrelax_solver, 279
- cp_close_schedule, 268
- cp_find_next_sol, 141
- cp_get_default_schedule_strategy, 258
- cp_get_linrelax, 272
- cp_infeas_analysis, 154
- cp_local_optimize, 158
- cp_maximise, 143
- cp_maximize, 144
- cp_minimise, 145
- cp_minimize, 146
- cp_post, 139
- cp_propagate, 140
- cp_remove_linrelax_solver, 278
- cp_reset_params, 160
- cp_reset_search, 142
- cp_restore_state, 156
- cp_save_state, 155
- cp_schedule, 231
- cp_set_branch_callback, 203
- cp_set_branching, 189
- cp_set_node_callback, 200
- cp_set_schedule_strategy, 257
- cp_set_solution_callback, 197
- cp_shave, 153

- cp_show_best_sol, 150
- cp_show_prob, 148
- cp_show_relax, 276
- cp_show_schedule, 215
- cp_show_sol, 149
- cp_show_stats, 192
- cp_show_var, 137
- cpctr, 7
- cpfloatvar, 5
- cpinlctr, 67
- cpnlinctr, 75
- cpresource, 14
- cptask, 14
- cpvar, 5
- create, 5, 17
- CSP, see constraint satisfaction problem
- cumulative, 101
- cycle, 85

D

- disjunction selection heuristic, 12
- disjunctive, 103
- distance, 79
- distribute, 99
- domain, 1

E

- Earliest Completion Time, 15
- Earliest Start Time, 15
- ECT, see Earliest Completion Time
- element, 94
- equiv, 71
- EST, see Earliest Start Time
- exp, 78
- export_prob, 285

F

- fix_to_relaxed, 274

G

- generate_cuts, 280
- generic_binary_constraint, 96
- generic_nary_constraint, 98
- get_indicator, 283
- get_linrelax, 284
- get_linrelax_solver, 273
- get_reduced_cost, 286
- get_relaxed_value, 287
- getactivebranch, 115
- getarity, 109
- getassignment, 262
- getcapacity, 252

getconsumption, 219
 getdegree, 123
 getduration, 218
 getelt, 165
 getend, 217
 getlb, 117
 getmakespan, 232
 getmiddle, 119
 getname, 151
 getnext, 126
 getprev, 127
 getpriority, 110
 getproduction, 223
 getprovision, 225
 getrand, 125
 getrequirement, 221
 getsetuptime, 255
 getsizes, 120, 166
 getsol, 147
 getstart, 216
 gettag, 112, 195
 gettarget, 124
 getub, 118
 getval, 121, 163
 group_serializer, 194

H

has_assignment, 264
 hybrid model, 25

I

implies, 73
 indicator variable, 22
 is_consuming, 227
 is_equal, 129
 is_fixed, 122, 208, 256
 is_idletime, 267
 is_producing, 229
 is_providing, 230
 is_requiring, 228
 is_same, 130

K

KALIS_ALL_CONSTRAINTS, 50
 KALIS_AUTO_PROPAGATE, 57
 KALIS_AUTO_RELAX, 64
 KALIS_BACKTRACKS, 59
 KALIS_BARRIER, 54
 KALIS_BILEVEL_RELAX_SOLVER, 49
 KALIS_CHECK_SOLUTION, 62
 KALIS_COMPUTATION_TIME, 59
 KALIS_COPYRIGHT, 40
 KALIS_DEFAULT_CONTINUOUS_LB, 62
 KALIS_DEFAULT_CONTINUOUS_UB, 62
 KALIS_DEFAULT_LB, 57
 KALIS_DEFAULT_PRECISION_RELATIVITY, 63
 KALIS_DEFAULT_PRECISION_VALUE, 63
 KALIS_DEFAULT_SCHEDULE_HORIZ_MAX, 63
 KALIS_DEFAULT_SCHEDULE_HORIZ_MIN, 63
 KALIS_DEFAULT_UB, 57

KALIS_DEPTH, 58
 KALIS_DICHOTOMIC_OBJ_SEARCH, 64
 KALIS_DISCRETE_RESOURCE, 40
 KALIS_DISJ_INPUT_ORDER, 44
 KALIS_DISJ_PRIORITY_ORDER, 44
 KALIS_DISJUNCTIONS, 45
 KALIS_DISTANCE_CONSTRAINTS, 50
 KALIS_DUAL_SIMPLEX, 54
 KALIS_EDGE_FINDING, 45
 KALIS_FORWARD_CHECKING, 31
 KALIS_GEN_ARC_CONSISTENCY, 32
 KALIS_INITIAL_SOLUTION, 46
 KALIS_INPUT_ORDER, 35
 KALIS_LARGEST_ECT, 42
 KALIS_LARGEST_EST, 42
 KALIS_LARGEST_LCT, 42
 KALIS_LARGEST_LST, 43
 KALIS_LARGEST_MAX, 36
 KALIS_LARGEST_MIN, 35
 KALIS_LARGEST_REDUCED_COST, 270
 KALIS_LINEAR_CONSTRAINTS, 49
 KALIS_LOGICAL_CONSTRAINTS, 50
 KALIS_MAX_BACKTRACKS, 60
 KALIS_MAX_COMPUTATION_TIME, 61
 KALIS_MAX_DEGREE, 35
 KALIS_MAX_DEPTH, 60
 KALIS_MAX_NODES, 59
 KALIS_MAX_NODES_BETWEEN_SOLUTIONS, 61
 KALIS_MAX_SOLUTIONS, 60
 KALIS_MAX_TO_MIN, 39
 KALIS_MAXIMIZE, 51
 KALIS_MAXREGRET_LB, 36
 KALIS_MAXREGRET_UB, 36
 KALIS_MIDDLE_VALUE, 39
 KALIS_MIN_TO_MAX, 39
 KALIS_MINIMIZE, 51
 KALIS_NARY_OBJ_SEARCH, 65
 KALIS_NEAREST_RELAXED_VALUE, 271
 KALIS_NEAREST_VALUE, 39
 KALIS_NETWORK_SIMPLEX, 55
 KALIS_NODES, 58
 KALIS_NON_LINEAR_CONSTRAINTS, 49
 KALIS_OPT_ABS_TOLERANCE, 61
 KALIS_OPT_REL_TOLERANCE, 61
 KALIS_OPTIMAL_SOLUTION, 46
 KALIS_OPTIMIZE_WITH_RESTART, 58
 KALIS_PRIMAL_SIMPLEX, 54
 KALIS_RANDOM_VALUE, 40
 KALIS_RANDOM_VARIABLE, 37
 KALIS_RELAX_ALGORITHM, 53
 KALIS_RELAX_GLOBAL, 52
 KALIS_RELAX_MIP_ABS_STOP, 53
 KALIS_RELAX_MIP_REL_STOP, 53
 KALIS_RELAX_OPT_TOL, 53
 KALIS_RELAX_PRESOLVE, 52
 KALIS_RELAX_RCOSTS_PROPAG, 52
 KALIS_RELAX_RELOAD_BASIS, 52
 KALIS_RESET_OPT_PARAMS, 48
 KALIS_RESET_PARAMS_ALL, 46
 KALIS_RESET_SEARCH_PARAMS, 48

KALIS_RESET_VAR_BOUNDS, 47
 KALIS_RESET_VAR_PRECISION, 47
 KALIS_SCHEDULE_ENABLE_SHAVING, 65
 KALIS_SDOMDEG_RATIO, 37
 KALIS_SEARCH_LIMIT, 58
 KALIS_SLIM_BY_BACKTRACKS, 33
 KALIS_SLIM_BY_DEPTH, 33
 KALIS_SLIM_BY_NODES, 32
 KALIS_SLIM_BY_SOLUTIONS, 33
 KALIS_SLIM_BY_TIME, 33
 KALIS_SLIM_UNREACHED, 32
 KALIS_SMALLEST_DOMAIN, 37
 KALIS_SMALLEST_ECT, 41
 KALIS_SMALLEST_EST, 41
 KALIS_SMALLEST_LCT, 41
 KALIS_SMALLEST_LST, 42
 KALIS_SMALLEST_MAX, 38
 KALIS_SMALLEST_MIN, 38
 KALIS_SOLVE_AS_LP, 51
 KALIS_SOLVE_AS_MIP, 51
 KALIS_TASK_INPUT_ORDER, 43
 KALIS_TASK_INTERVALS, 45
 KALIS_TASK_RANDOM_ORDER, 43
 KALIS_THREADS, 65
 KALIS_TIMETABLING, 44
 KALIS_TLIM_ABS_OPT, 34
 KALIS_TLIM_REL_OPT, 34
 KALIS_TLIM_UNREACHED, 34
 KALIS_TOLERANCE_LIMIT, 59
 KALIS_TOPNODE_RELAX_SOLVER, 48
 KALIS_TREENODE_RELAX_SOLVER, 49
 KALIS_UNARY_RESOURCE, 40
 KALIS_USE_3B_CONSISTENCY, 64
 KALIS_VERBOSE_LEVEL, 64
 KALIS_WIDEST_DOMAIN, 38

L

Latest Completion Time, 15
 Latest Start Time, 15
 LCT, see Latest Completion Time
 linear relaxation, 22
 branching, 24
 ln, 77
 lp_optimize, 281
 LST, see Latest Start Time

M

Mathematical Programming, 21
 maximum_minimum, 90
 meta branching scheme, 194

O

occurrence, 92
 optimization, 12
 or, 70

P

path_order, 157
 probe_assign_var, 178
 probe_settle_disjunction, 179
 producer_consumer, 105

produces, 239
 provides, 242

R

relaxation variable, 22
 requires, 236
 resusage, 259

S

search techniques, 2
 set_integer, 288
 set_linrelax_solver_attribute, 282
 set_resource_attributes, 245
 set_reversible_attributes, 161
 set_sol_as_target, 159
 set_task_attributes, 209
 set_verbose_level, 275
 setcapacity, 246
 setdomain, 132
 setduration, 210
 setelt, 164
 setfirstbranch, 114
 setidletimes, 266
 setlb, 133
 setmaxavailability, 248
 setminusage, 250
 setname, 152
 setprecision, 136
 setpredecessors, 212
 setpriority, 111
 setsetuptime, 254
 setsuccessors, 214
 settag, 113
 settarget, 131
 settle_disjunction, 176
 setub, 134
 setval, 135, 162
 solution, 2
 split_domain, 181

T

task_serialize, 185
 tree search, 2

V

value selection heuristic, 12
 variable selection heuristic, 11