

Xpress-Kalis

User guide

Release 2008

Published by Fair Isaac Corporation

© Copyright Fair Isaac Corporation and Artelys SA 2008. All rights reserved.

All trademarks referenced in this manual that are not the property of Fair Isaac or Artelys SA are acknowledged.

All companies, products, names and data contained within this document are completely fictitious and are used solely to illustrate the use of Xpress-MP and Kalis. Any similarity between these names or data and reality is purely coincidental.

How to Contact Fair Isaac

USA, Canada and all Americas

Information and Sales: info@dashoptimization.com

Licensing: license-usa@dashoptimization.com

Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445

Fax: +1 (201) 567 9443

Fair Isaac
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan

Information and Sales: xpress@msi-jp.com

Licensing: xpress@msi-jp.com

Product Support: xpress@msi-jp.com

Tel: +81 43 297 8841

Fax: +81 43 297 8836

MSI CO., LTD
WBG Marive-West Tower 2
2-6 Nakase Mihama-ku
261-7102 Chiba
Japan

Worldwide

Information and Sales: info@dashoptimization.com

Licensing: license@dashoptimization.com

Product Support: support@dashoptimization.com

Tel: +44 1926 315862

Fax: +44 1926 315854

Fair Isaac
Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at <http://www.dashoptimization.com> 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	Xpress-Kalis	1
1.1.1	Note on product versions	2
1.2	Software installation	2
1.3	Basic concepts of Constraint Programming	2
1.4	Contents of this document	3
I	Working with Xpress-Kalis	4
2	Modeling basics	5
2.1	A first model	5
2.1.1	Implementation with Mosel	5
2.1.1.1	General structure	6
2.1.1.2	Solving and solution output	6
2.1.1.3	Formatting	7
2.1.2	Running the model	7
2.1.2.1	Working from the Mosel command line	7
2.1.2.2	Using IVE	8
2.1.2.3	Debugging a model	9
2.2	Data input from file	10
2.2.1	Model formulation	10
2.2.2	Implementation	11
2.2.3	Results	11
2.2.4	Data-driven model	12
2.3	Optimization and enumeration	13
2.3.1	Optimization	13
2.3.2	Enumeration	14
2.3.2.1	IVE search tree display	16
2.4	Continuous variables	16
3	Constraints	18
3.1	Constraint handling	18
3.1.1	Model formulation	19
3.1.2	Implementation	19
3.1.3	Results	19
3.1.4	Naming constraints	20
3.1.5	Explicit posting of constraints	21
3.1.6	Explicit constraint propagation	21
3.2	Arithmetic constraints	21
3.3	all_different: Sudoku	22
3.3.1	Model formulation	22
3.3.2	Implementation	23
3.3.3	Results	25
3.4	abs and distance: Frequency assignment	25
3.4.1	Model formulation	26
3.4.2	Implementation	26
3.4.3	Results	28

3.5	element: Sequencing jobs on a single machine	29
3.5.1	Model formulation 1	29
3.5.2	Implementation of model 1	30
3.5.3	Results	32
3.5.4	Alternative formulation using disjunctions	33
3.5.5	Implementation of model 2	34
3.6	occurrence: Sugar production	36
3.6.1	Model formulation	36
3.6.2	Implementation	36
3.6.3	Results	38
3.7	distribute: Personnel planning	38
3.7.1	Model formulation	38
3.7.2	Implementation	38
3.7.3	Results	40
3.8	implies: Paint production	41
3.8.1	Formulation of model 1	41
3.8.2	Implementation of model 1	42
3.8.3	Formulation of model 2	43
3.8.4	Implementation of model 2	43
3.8.5	Results	44
3.9	equiv: Location of income tax offices	44
3.9.1	Model formulation	45
3.9.2	Implementation	46
3.9.3	Results	47
3.10	cycle: Paint production	47
3.10.1	Model formulation	48
3.10.2	Implementation	48
3.10.3	Results	49
3.11	Generic binary constraints: Euler knight tour	49
3.11.1	Model formulation	49
3.11.2	Implementation	50
3.11.3	Results	51
3.11.4	Alternative implementation	51
3.11.5	Alternative implementation 2	54
4	Enumeration	56
4.1	Predefined search strategies	56
4.2	Interrupting and restarting the search	57
4.3	Callbacks	58
4.4	User-defined enumeration strategies	58
4.4.1	Model formulation	59
4.4.1.1	Parallel machine assignment	59
4.4.1.2	Machines working in series	59
4.4.2	Implementation	59
4.4.3	User search	61
4.4.4	Results	63
5	Scheduling	64
5.1	Tasks and resources	64
5.2	Precedences	65
5.2.1	Model formulation	65
5.2.2	Implementation	66
5.2.3	Results	67
5.2.4	Alternative formulation without scheduling objects	67
5.3	Disjunctive scheduling: unary resources	68
5.3.1	Model formulation	68
5.3.2	Implementation	69
5.3.3	Results	71
5.4	Cumulative scheduling: discrete resources	71

5.4.1	Model formulation	72
5.4.2	Implementation	72
5.4.3	Results	73
5.4.4	Alternative formulation without scheduling objects	74
5.4.5	Implementation	74
5.5	Renewable and non-renewable resources	75
5.5.1	Model formulation	76
5.5.2	Implementation	76
5.5.3	Results	78
5.5.4	Alternative formulation without scheduling objects	79
5.5.5	Implementation	79
5.6	Extensions: setup times	81
5.6.1	Model formulation	81
5.6.2	Implementation	81
5.6.3	Results	82
5.7	Enumeration	83
5.7.1	Variable-based enumeration	83
5.7.1.1	Using <code>cp_minimize</code>	83
5.7.1.2	Using <code>cp_schedule</code>	84
5.7.2	Task-based enumeration	84
5.7.2.1	Model formulation	85
5.7.2.2	Implementation	85
5.7.2.3	Results	88
5.7.2.4	Alternative search strategies	89
5.7.3	Choice of the propagation algorithm	90
II	Xpress-Kalis extensions	91
6	Implementing Kalis extensions	92
6.1	Software architecture	92
6.2	Required software and installation	93
6.3	Compilation	93
7	Writing a user constraint	95
7.1	Model formulation	95
7.2	Implementation: Mosel model	96
7.3	Implementation: user extension	97
7.3.1	List of Mosel subroutines	97
7.3.2	Xpress-Kalis interface functions	98
7.3.3	Implementation of the Mosel function	99
7.3.4	The Kalis user constraint class	100
7.3.5	Improving the constraint propagation algorithm	103
8	Writing a user branching scheme	105
8.1	Model formulation	105
8.2	Implementation: Mosel model	106
8.3	Implementation: user extension	108
8.3.1	List of Mosel subroutines	108
8.3.2	Xpress-Kalis interface functions	109
8.3.3	Implementation of the Mosel function	109
8.3.4	The Kalis value selection strategy class	111
8.4	Implementing a complete branching scheme	113
8.4.1	The Kalis variable selection strategy class	113
8.4.2	The Kalis branching scheme class	114

Appendix	117
A Trouble shooting	118
B Glossary of CP terms	119
 Bibliography	 121
 Index	 122

Chapter 1

Introduction

Constraint Programming is an approach to problem solving that has been particularly successful for dealing with nonlinear constraint relations over discrete variables. In the following we therefore often use ‘Constraint Programming’ or ‘CP’ synonymously to ‘finite domain Constraint Programming’.

In the past, CP has been successfully applied to such different problem types as production scheduling (with or without resource constraints), sequencing and assignment of tasks, workforce planning and timetabling, frequency assignment, loading and cutting, and also graph coloring.

The strength of CP lies in its use of a high-level semantics for stating the constraints that preserves the original meaning of the constraint relations (such high-level constraints are referred to as *global constraints*). It is not necessary to translate constraints into an arithmetic form—a process whereby sometimes much of the problem structure is lost. The knowledge about a problem inherent to the constraints is exploited by the solution algorithms, rendering them more efficient.

1.1 Xpress-Kalis

Xpress-Kalis, or *Kalis for Mosel*, provides access to the Artelys Kalis[©] Constraint Programming solver from a Mosel module, *kalis*. Through Xpress-Kalis, the Constraint Programming functionality of Kalis becomes available in the Mosel environment, allowing the user to formulate and solve CP models in the Mosel language. Xpress-Kalis combines a finite domain solver and a solver over continuous (floating point) variables. To aid the formulation of scheduling problems, the software defines specific aggregate modeling objects representing tasks and resources that will automatically setup certain (implicit) constraint relations and trigger the use of built-in search strategies specialized for this type of problem. Standard scheduling problems may thus be defined and solved simply by setting up the corresponding task and resource objects.

All data handling facilities of the Mosel environment, including data transfer in memory (using the Mosel IO drivers) and ODBC access to databases (through the module *mmodbc*) can be used with *kalis* without any extra effort.

The Mosel language supports typical programming constructs, such as loops, subroutines, etc., that may be required to implement more complicated algorithms. Mosel can also be used as a platform for combining different solvers, in particular Xpress-Kalis with Xpress-Optimizer for joint CP – LP/MIP problem solving¹. This manual explains the basics on modeling and programming with Mosel and, where necessary, also some more advanced features. For a complete documentation and a more thorough introduction to its use the reader is referred to the [Mosel language reference manual](#) and the [Mosel user guide](#).

¹See the Xpress-MP Whitepaper [Multiple models and parallel solving with Mosel](#)

Beyond the functionality readily available from Xpress-Kalis the software also provides a unique extension mechanism that opens it up to various kinds of additions of new functionality on the library level that become available within the Mosel language through the Mosel Native Interface (see the [Mosel NI reference manual](#) and the [Mosel NI user guide](#)). Such user-defined extensions principally relate to the definition of new constraints and branching schemes.

1.1.1 Note on product versions

The examples in this manual have been developed using the release 2007.1.0 of Xpress-Kalis with the Xpress-MP Release 2007A beta version of Mosel (1.7.9) and version 1.17.50 of Xpress-IVE. If they are run with other product versions the output obtained may look different. In particular, improvements to the algorithms in the CP solver or modifications to the default settings in Xpress-Kalis may influence the behavior of the constraints or the search. The IVE interface may also undergo slight changes in future releases as new features are added.

1.2 Software installation

To be able to work with Xpress-Kalis, the Xpress-Mosel software and Xpress-Kalis must be installed and licensed. Windows users may find it convenient to install (in addition) the graphical environment Xpress-IVE for working with CP models, but this is not a prerequisite.

Follow the installation instructions provided with the Xpress-MP distribution for installing Mosel (and IVE) on your computer. Then install Xpress-Kalis according to the instructions provided with the Xpress-Kalis distribution.

1.3 Basic concepts of Constraint Programming

A *Constraint Programming (CP) problem* is defined by its *decision variables* with their *domains* and *constraints* over these variables. The problem definition is usually completed by a *branching strategy* (also referred to as *enumeration* or *search strategy*).

CP makes active use of the concept of *variable domains*, that is, the set out of which a decision variable takes its values. In *finite domain Constraint Programming* these are sets or intervals of integer numbers.

Each *constraint* in CP comes with its own (set of) solution algorithm(s), typically based on results from other areas, such as graph theory. Once a constraint has been established it maintains its set of variables in a solved state, *i.e.*, its solution algorithm removes any values that it finds infeasible from the domains of the variables.

The constraints in a CP problem are linked by a mechanism called *constraint propagation*: whenever the domain of a variable is modified this triggers a re-evaluation of all constraints on this variable which in turn may cause modifications to other variables or further reduction of the domain of the first variable as shown in the example in Figure 1.1 (the original domains of the variables are reduced by the addition of two constraints; in the last step the effect of the second constraint is propagated to the first constraint, triggering its re-evaluation).

A CP problem is built up *incrementally* by adding constraints and bounds on its variables. The solving of a CP problem starts with the statement of the first constraint—values that violate the constraint relation are removed from the domains of the involved variables. Since the effect of a newly added constraint is propagated immediately to the entire CP problem it is generally not possible to modify or delete this constraint from the problem later on.

In some cases the combination of constraint solving and the propagation mechanism may be sufficient to prove that a problem instance is infeasible, but most of the time it will be necessary to add an *enumeration* for reducing all variable domains to a single value (*consistent instantiation* or *feasible solution*) or proving that no such solution exists. In addition it is possible to define an *objective function* (or *cost function*) and search for a feasible solution with the best objective function value (*optimal solution*).

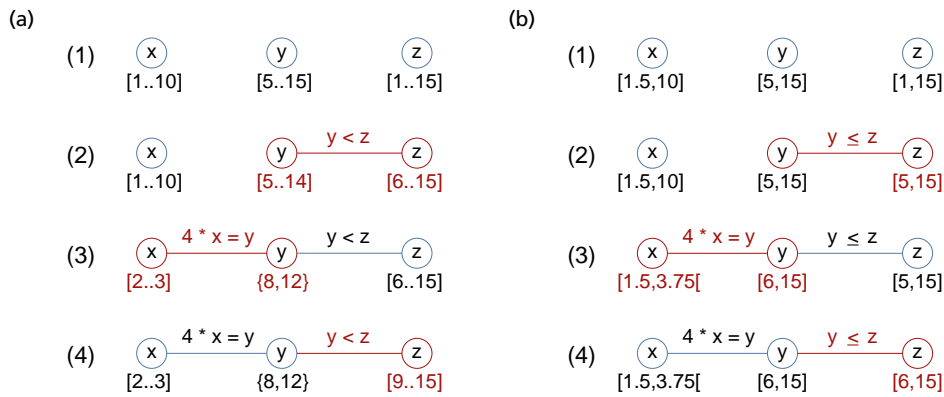


Figure 1.1: Example of constraint propagation, (a) finite domain (discrete) variables, (b) continuous variables.

1.4 Contents of this document

This document gives an introduction to working with Xpress-Kalis. Basic notions of Constraint Programming are explained but it is expected that the reader has some general understanding of CP techniques. Since Xpress-Kalis is a module for Xpress-Mosel this document also describes features of the Mosel language where this appears necessary to the understanding of the presented examples.

By means of example models we illustrate the use of the new types defined by Xpress-Kalis, namely

- Decision variables: finite domain and floating point variables
- Constraints: absolute value and distance, all-different, element, generic binary, linear, maximum and minimum, occurrence, and logical relations
- Enumeration: predefined branching schemes and user search strategies
- Scheduling: aggregate modeling objects representing tasks and resources

The first chapter deals with the basics of writing CP models with Mosel. It explains the general structure of CP models and some basics on working with Mosel, including data handling. Then follows a series of small problems illustrating the use of the different constraint relation types in Xpress-Kalis. The next chapter introduces in a more systematic way the different possibilities of defining enumeration strategies, some of which already appear in the preceding model examples. The chapter dedicated to the topic of scheduling introduces the modeling objects 'task' and 'resource' that simplify the formulation of scheduling problems.

The last part of this document is concerned with the implementation of Xpress-Kalis extensions. This part is aimed at expert users who wish to implement their own constraint relations or new branching schemes. The unique extension mechanism of Xpress-Kalis makes new functionality implemented on the library level available within the Mosel language through the Mosel Native Interface.

Apart from the initial examples, every example is presented as follows:

1. Example description
2. Formulation as a CP model
3. Implementation with Xpress-Kalis: code listing and explanations
4. Results

All example models of this document are included with the set of examples that is provided as part of the Xpress-Kalis distribution.

I. Working with Xpress-Kalis

Chapter 2

Modeling basics

This chapter shows how to

- start working with Mosel,
- create and solve a simple CP model with Xpress-Kalis,
- understand and analyze the output produced by the software,
- extend the model with data handling,
- define an objective function, and
- modify the default branching strategy.

2.1 A first model

Consider the following problem: we wish to schedule four meetings A, B, C, and D in three time slots (numbered 1 to 3). Some meetings are attended by the same persons, meaning that they may not take place at the same time: meeting A cannot be held at the same time as B or D, and meeting B cannot take the same time slot as C or D.

More formally, we may write down this problem as follows, where $plan_m$ ($m \in MEETINGS = \{A, B, C, D\}$) denotes the time slot for meeting m —these are the *decision variables* of our problem.

$$\begin{aligned} \forall m \in MEETINGS : plan_m &\in \{1, 2, 3\} \\ plan_A &\neq plan_B \\ plan_A &\neq plan_D \\ plan_B &\neq plan_C \\ plan_B &\neq plan_D \end{aligned}$$

2.1.1 Implementation with Mosel

The following code listing implements and solves the problem described above.

```
model "Meeting"
uses "kalis"

declarations
  MEETINGS = {'A', 'B', 'C', 'D'}           ! Set of meetings
  TIME = 1..3                               ! Set of time slots
  plan: array(MEETINGS) of cpvar           ! Time slot per meeting
end-declarations

forall(m in MEETINGS) setdomain(plan(m), TIME)
```

```

! Respect incompatibilities
plan('A') <> plan('B')
plan('A') <> plan('D')
plan('B') <> plan('C')
plan('B') <> plan('D')

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

! Solution printing
forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model

```

This Mosel model is saved as a text file with the name `meeting.mos`. Let us now take a closer look at what we have just written.

2.1.1.1 General structure

Every Mosel program starts with the keyword `model`, followed by a model name chosen by the user. The Mosel program is terminated with the keyword `end-model`.

As Mosel is itself not a solver, we specify that the Kalis constraint solver is to be used with the statement

```
uses "kalis"
```

at the begin of the model.

All objects must be declared in a `declarations` section, unless they are defined unambiguously through an assignment. For example, `i := 1` defines `i` as an integer and assigns to it the value 1. There may be several such `declarations` sections at different places in a model.

In the present case, we define two sets, and one array:

- `MEETINGS` is a *set of strings*.
- `TIME` is a so-called *range set*—i.e., a set of consecutive integers (here: from 1 to 3).
- `plan` is an array of decision variables of type `cpvar` (finite domain CP variables; a second decision variable type of Xpress-Kalis is `cpfloatvar` for continuous variables), indexed by the set `MEETINGS`.

The model then defines the domains of the variables using the Xpress-Kalis procedure `set-domain`. The decision variables are indeed created at their declaration with a large default domain and `setdomain` reduces these domains to the intersection of the default domain with the indicated values. As in the mathematical model, we use a `forall` loop to enumerate all the indices in the set `MEETINGS`.

This is followed by the statement of the constraints, in this model we have four disequality constraints.

2.1.1.2 Solving and solution output

With the function `cp_find_next_sol`, we call Kalis to solve the problem (find a feasible assignment of values to all decision variables). We test the return value of this function: if no solution is found it returns `false` and we stop the model execution at this point (by calling the Mosel procedure `exit`), otherwise we print out the solution.

To solve the problem Xpress-Kalis uses its built-in default search strategies. We shall see later how to modify these strategies.

The solution for a CP variable is obtained with the function `getsol`. To write several items on a single line use `write` instead of `writeln` for printing the output.

2.1.1.3 Formatting

Indentation, spaces, and empty lines in our model have been added to increase readability. They are skipped by Mosel.

Line breaks: It is possible to place several statements on a single line, separating them by semicolons, as such:

```
plan('A') <> plan('B'); plan('A') <> plan('D')
```

But since there are no special 'line end' or continuation characters, every line of a statement that continues over several lines must end with an operator (+, >=, etc.) or characters like ',' that make it obvious that the statement is not terminated.

As shown in the example, single line *comments* in Mosel are preceded by `!`. Multiple line comments start with `(!` and terminate with `!)`.

2.1.2 Running the model

You may choose among three different methods for running your Mosel models:

1. From the *Mosel command line*: this method can be used on all platforms for which Mosel is available. It is especially useful if you wish to execute a (batch) sequence of model runs—for instance, with different parameter settings. The full Mosel functionality, including its debugger, is accessible in this run mode.
2. Within the graphical environment *Xpress-IVE*: available to Windows users. IVE is a complete modeling and optimization development environment with a built-in text editor for working with Mosel models and a number of solution and search tree displays that help analyze models and solution algorithms in the development phase. Models can be modified and re-run interactively.
3. From within an *application program*: Mosel models may be executed and accessed from application programs (C/C++, Java, VB, .NET). This functionality is typically used for the deployment of Mosel models, integrating them into a company's information system.

In this manual we shall use the first two methods for running the models we develop. For further detail on embedding models into application programs the user is referred to the [Mosel user guide](#).

2.1.2.1 Working from the Mosel command line

When you have entered the complete model into the file `meeting.mos`, we can proceed to the solution to our problem. Three stages are required:

1. Compiling `meeting.mos` to a compiled file, `meeting.bim`
2. Loading the compiled file `meeting.bim`
3. Running the model we have just loaded.

We start Mosel at the command prompt, and type the following sequence of commands

```
mosel
compile meeting
load meeting
run
quit
```

which will compile, load and run the model. We will see output something like that below, where we have highlighted Mosel's output in bold face.

```
mosel
** Xpress-Mosel **
(c) Copyright Fair Isaac Corporation 2008
>compile meeting
Compiling 'meeting'...
>load meeting
>run
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
Returned value: 0
>quit
Exiting.
```

Since the compile/load/run sequence is so often used, it can be abbreviated to

```
exec meeting
```

The same steps may be done immediately from the command line:

```
mosel -c "compile meeting; load meeting; run"
```

or

```
mosel -c "exec meeting"
```

The `-c` option is followed by a list of commands enclosed in double quotes.

2.1.2.2 Using IVE

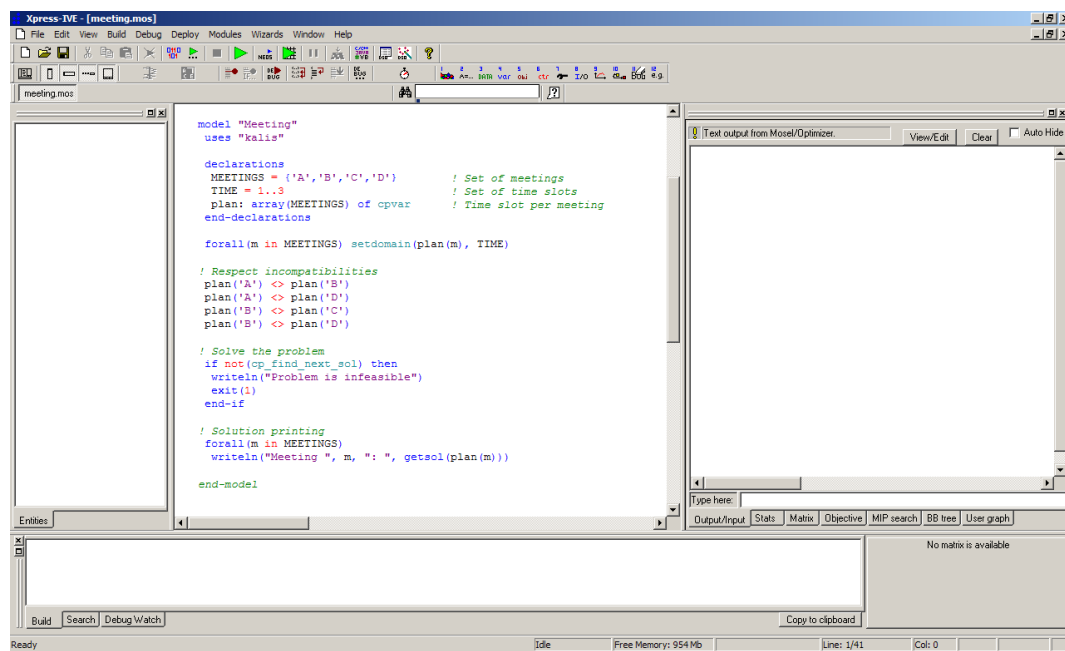



Figure 2.1: IVE after opening a model

To execute the model file `meeting.mos` with IVE you need to carry out the following steps.

- Start up IVE: if you have followed the standard installation procedure for Xpress-IVE, start the program by double clicking the icon on the desktop or selecting *Start* » *Programs* » *Xpress-MP* » *Xpress-IVE*. Otherwise, you may also start up IVE by typing `ive` in a DOS command prompt or by double clicking a model file (file with extension `.mos`).
- Open the model file by choosing *File* » *Open*. The model source is then displayed in the central window (the *IVE Editor*).
- Click the *Run* button  or, alternatively, choose *Build* » *Run*.

The *Build* pane at the bottom of the workspace displays the model execution status messages from Mosel. If syntax errors are found in the model they are displayed here, with details of the line and character position where the error was detected and a description of the problem, if available. Clicking on the error takes the user to the offending line.

When a model is run, the *Output/Input* pane at the right hand side of the workspace displays any output generated by the program. IVE will also provide a graphical representation of the CP search tree (*CP search* pane) and display summary problem statistics (*CP stats* pane). IVE also allows the user to draw graphs by embedding subroutines in Mosel models (see the documentation of module *mmive* for further detail).

IVE makes all information about the solution available through the *Entities* pane in the left hand window. By expanding the list of decision variables in this pane and hovering over one with the mouse pointer, its solution value is displayed.

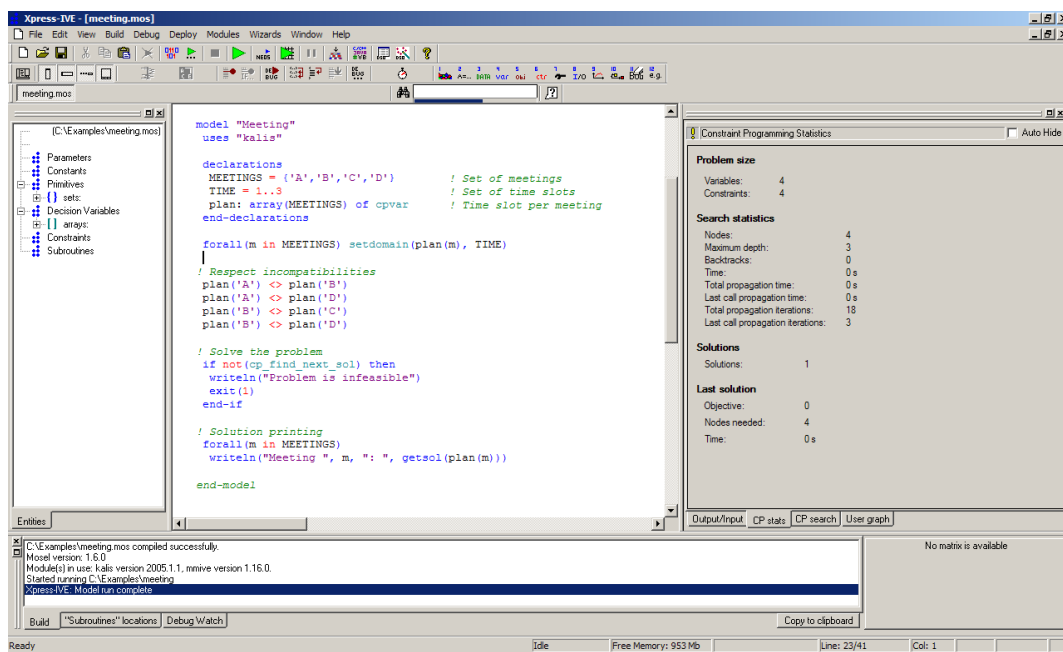


Figure 2.2: IVE after model execution

2.1.2.3 Debugging a model

A first step for debugging a model certainly is to add additional output. In our model, we could, for instance, print out the definition of the decision variables by adding the line

```
writeln(plan)
```

after the definition of the variables' domains, or even print out the complete problem definition with the procedure `cp_show_prob`. To obtain a more easily readable output for debug-

ging the user may give names to the decision variables of his problem. For example:

```
forall(m in MEETINGS) setname(plan(m), "Meeting "+m)
```


Notice that we have used the '+' sign to concatenate strings.

Calling the procedure `cp_show_stats` will display summary statistics of the CP solving.

To obtain detailed information about run-time errors with the command line version models need to be compiled with the flag `-g`, for example,

```
mosel -c "exec -g meeting"
```

For using the Mosel debugger (please see the [Mosel language reference manual](#) for further detail) the compilation flag `-G` needs to be used instead.

IVE by default compiles models in debug mode and re-compiles them correspondingly if the debugger is started (by clicking the button  or via the *Debug* menu).

2.2 Data input from file

We now extend the previous example in order to use it with different data sets. If we wish to run a model with different data sets, it would be impractical and error-prone having to edit the model file with every change of the data set. Instead, we are now going to see how to read data from a text file.

This is a description of the problem we want to solve (example taken from Section 14.4 of the book '[Applications of optimization with Xpress-MP](#)').

A technical university needs to schedule the exams at the end of the term for a course with several optional modules. Every exam lasts two hours. Two days have been reserved for the exams with the following time periods: 8:00–10:00, 10:15–12:15, 14:00–16:00, and 16:15–18:15, resulting in a total of eight time periods. For every exam the set of incompatible exams that may not take place at the same time because they have to be taken by the same students is shown in Table 2.1.

Table 2.1: Incompatibilities between different exams

	DA	NA	C++	SE	PM	J	GMA	LP	MP	S	DSE
DA	–	X	–	–	X	–	X	–	–	X	X
NA	X	–	–	–	X	–	X	–	–	X	X
C++	–	–	–	X	X	X	X	–	X	X	X
SE	–	–	X	–	X	X	X	–	–	X	X
PM	X	X	X	X	–	X	X	X	X	X	X
J	–	–	X	X	X	–	X	–	X	X	X
GMA	X	X	X	X	X	X	–	X	X	X	X
LP	–	–	–	–	X	–	X	–	–	X	X
MP	–	–	X	–	X	X	X	–	–	X	X
S	X	X	X	X	X	X	X	X	X	–	X
DSE	X	X	X	X	X	X	X	X	X	X	–

2.2.1 Model formulation

The CP model has the same structure as the previous one, with the difference that we now introduce a data array *INCOMP* indicating incompatible pairs of exams and define the disequality constraints in a loop instead of writing them out one by one.

$$\begin{aligned} \forall e \in EXAM : plan_e &\in \{1, \dots, 8\} \\ \forall d, e \in EXAM, INCOMP_{de} = 1 : plan_d &\neq plan_e \end{aligned}$$

2.2.2 Implementation

The Mosel model now looks as follows.

```
model "I-4 Scheduling exams (CP) "
uses "kalis"

declarations
  EXAM = 1..11                                ! Set of exams
  TIME = 1..8                                  ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams
  EXAMNAME: array(EXAM) of string

  plan: array(EXAM) of cpvar                  ! Time slot for exam
end-declarations

EXAMNAME:: (1..11) ["DA", "NA", "C++", "SE", "PM", "J", "GMA", "LP", "MP", "S", "DSE"]

initializations from 'Data/i4exam.dat'
  INCOMP
end-initializations

forall(e in EXAM) setdomain(plan(e), TIME)

! Respect incompatibilities
forall(d,e in EXAM | d<e and INCOMP(d,e)=1)  plan(d) <> plan(e)

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

! Solution printing
forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM)
    if (getsol(plan(e))=t) then write(EXAMNAME(e), " "); end-if
  writeln
end-do

end-model
```

The values of the array *INCOMP* are read in from the file *i4exam.dat* in an *initializations* block. In the definition of the disequality constraints we check the value of the corresponding array entry—conditions on the indices for loops, sums, and other aggregate operators are marked by a vertical bar.

The data file has the following contents.

```
INCOMP: [0 1 0 0 1 0 1 0 0 1 1
         1 0 0 0 1 0 1 0 0 1 1
         0 0 0 1 1 1 1 0 1 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 0 1 1 1 1 1 1
         0 0 1 1 1 0 1 0 1 1 1
         1 1 1 1 1 1 0 1 1 1 1
         0 0 0 0 1 0 1 0 0 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 1 1 1 1 1 0 1
         1 1 1 1 1 1 1 1 1 0]
```

2.2.3 Results

The model prints out the following results. Only the first seven time slots are used for scheduling exams.

```
Slot 1: DA C++ LP
Slot 2: NA SE MP
```

```

Slot 3: PM
Slot 4: GMA
Slot 5: S
Slot 6: DSE
Slot 7: J
Slot 8:

```

2.2.4 Data-driven model

In the model shown above, we have read the incompatibility data from a file but part of the data (namely the index set *EXAM* and the number of time slots available) are still hard-coded in the model. To obtain a fully flexible model that can be run with arbitrary data sets we need to move all data definitions from the model to the data file.

The new data file `i4exam2.dat` not only defines the data entries, it also defines the index tuples for the array *INCOMP*. Every data entry is preceded by its index tuple (in brackets). There is no need to write out explicitly the contents of the set *EXAM*—Mosel will automatically populate this set with the index values read in for the array *INCOMP*. In addition, the data file now contains a value for the number of time periods *NT*.

```

INCOMP: [ ("DA" "NA") 1 ("DA" "PM") 1 ("DA" "GMA") 1 ("DA" "S") 1 ("DA" "DSE") 1
          ("NA" "DA") 1 ("NA" "PM") 1 ("NA" "GMA") 1 ("NA" "S") 1 ("NA" "DSE") 1
          ("C++" "SE") 1 ("C++" "PM") 1 ("C++" "J") 1 ("C++" "GMA") 1
          ("C++" "MP") 1 ("C++" "S") 1 ("C++" "DSE") 1
          ("SE" "C++") 1 ("SE" "PM") 1 ("SE" "J") 1 ("SE" "GMA") 1 ("SE" "S") 1
          ("SE" "DSE") 1
          ("PM" "DA") 1 ("PM" "NA") 1 ("PM" "C++") 1 ("PM" "SE") 1 ("PM" "J") 1
          ("PM" "GMA") 1 ("PM" "LP") 1 ("PM" "MP") 1 ("PM" "S") 1 ("PM" "DSE") 1
          ("J" "C++") 1 ("J" "SE") 1 ("J" "PM") 1 ("J" "GMA") 1 ("J" "MP") 1
          ("J" "S") 1 ("J" "DSE") 1
          ("GMA" "DA") 1 ("GMA" "NA") 1 ("GMA" "C++") 1 ("GMA" "SE") 1
          ("GMA" "PM") 1 ("GMA" "J") 1 ("GMA" "LP") 1 ("GMA" "MP") 1
          ("GMA" "S") 1 ("GMA" "DSE") 1
          ("LP" "PM") 1 ("LP" "GMA") 1 ("LP" "S") 1 ("LP" "DSE") 1
          ("MP" "C++") 1 ("MP" "PM") 1 ("MP" "J") 1 ("MP" "GMA") 1 ("MP" "S") 1
          ("MP" "DSE") 1
          ("S" "DA") 1 ("S" "NA") 1 ("S" "C++") 1 ("S" "SE") 1 ("S" "PM") 1
          ("S" "J") 1 ("S" "GMA") 1 ("S" "LP") 1 ("S" "MP") 1 ("S" "DSE") 1
          ("DSE" "DA") 1 ("DSE" "NA") 1 ("DSE" "C++") 1 ("DSE" "SE") 1
          ("DSE" "PM") 1 ("DSE" "J") 1 ("DSE" "GMA") 1 ("DSE" "LP") 1
          ("DSE" "MP") 1 ("DSE" "S") 1 ]

NT: 8

```

Our model also needs to undergo a few changes: the sets *EXAM* and *TIME* are now declared by stating their types, which turns them into *dynamic sets* (as opposed to their previous constant definition by stating their values). As a consequence, the array of decision variables *plan* is declared before the indexing set *EXAM* is known and Mosel creates this array as a *dynamic array*, meaning that the declaration results in an empty array and its elements need to be created explicitly (using the Mosel procedure `create`) once the indices are known. Before creating the variables, we modify the default bounds of Xpress-Kalis to the values corresponding to the set *TIME*, thus replacing the call to `setdomain`.

The declaration of array *INCOMP* results in a dynamic array as well. The `initializations` block will assign values to just those entries that are listed in the data file (with the previous, constant declaration, all entries were defined). This makes it possible to reformulate the condition on the loop defining the disequality constraints: we now simply test for the existence of an entry instead of comparing all data values. With larger data sets, using the keyword `exists` may greatly reduce the execution time of loops involving sparse arrays (multidimensional data arrays with few entries different from 0).

```

model "I-4 Scheduling exams (CP) - 2"
uses "kalis"

declarations
  NT: integer                ! Number of time slots
  EXAM: set of string        ! Set of exams

```

```

    TIME: set of integer                ! Set of time slots
    INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

    plan: array(EXAM) of cpvar          ! Time slot for exam
end-declarations

initializations from 'Data/i4exam2.dat'
    INCOMP NT
end-initializations

TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
forall(e in EXAM) create(plan(e))

! Respect incompatibilities
forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e) plan(d) <> plan(e)

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

! Solution printing
forall(t in TIME) do
    write("Slot ", t, ": ")
    forall(e in EXAM)
        if (getsol(plan(e))=t) then write(e, " "); end-if
    writeln
end-do

end-model

```

Running this fully data-driven model produces the same solution as the previous version.

An alternative to the explicit creation of the decision variables *plan* is to move their declaration after the initialization of the data as shown in the code extract below. In this case, it is important to `finalize` the indexing set *EXAM*, which turns it into a constant set with its current contents and allows Mosel to create any subsequently declared arrays indexed by this set as static arrays.

```

declarations
    NT: integer                ! Number of time slots
    EXAM: set of string        ! Set of exams
    TIME: set of integer       ! Set of time slots
    INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams
end-declarations

initializations from 'Data/i4exam2.dat'
    INCOMP NT
end-initializations

finalize(EXAM)
TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
declarations
    plan: array(EXAM) of cpvar          ! Time slot for exam
end-declarations

```

2.3 Optimization and enumeration

2.3.1 Optimization

Since running our model `i4exam_ka.mos` in Section 2.2.2 has produced a solution to the problem that does not use all time slots one might wonder which is the minimum number of time slots that are required for this problem. This question leads us to the formulation of an *opti-*

mization problem.

We introduce a new decision variable *numslot* over the same value range as the *plan_i* variables and add the constraints that this variable is greater or equal to every *plan_i* variable. A simplified formulation is to say that the variable *numslot* equals the *maximum value* of all *plan_i* variables.

The objective then is to minimize the value of *numslot*, which results in the following model.

```
model "I-4 Scheduling exams (CP) - 3"
uses "kalis"

declarations
  NT: integer                ! Number of time slots
  EXAM: set of string        ! Set of exams
  TIME: set of integer       ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar ! Time slot for exam
  numslot: cpvar             ! Number of time slots used
end-declarations

initializations from 'Data/i4exam2.dat'
  INCOMP NT
end-initializations

finalize(EXAM)
TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
forall(e in EXAM) create(plan(e))

! Respect incompatibilities
forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e) plan(d) <> plan(e)

! Calculate number of time slots used
numslot = maximum(plan)

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

! Solution printing
forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM)
    if (getsol(plan(e))=t) then write(e, " "); end-if
  writeln
end-do

end-model
```

Instead of `cp_find_next_sol` we now use `cp_minimize` with the objective function variable *numslot* as function argument.

This program also generates a solution using seven time slots, thus proving that this is the least number of slots required to produce a feasible schedule.

2.3.2 Enumeration

When comparing the problem statistics in IVE's *CP stats* pane or those obtained by adding a call to `cp_show_stats` to the end of the different versions of our model, we can see that switching from finding a feasible solution to optimization considerably increases the number of nodes explored by the CP solver.

So far we have simply relied on the default enumeration strategies of Xpress-Kalis. We shall now try to see whether we can reduce the number of nodes explored and hence shorten the time spent by the search for proving optimality.

The default strategy of Kalis for enumerating finite domain variables corresponds to adding the statement

```
cp_set_branching(assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))
```

before the start of the search. `assign_var` denotes the *branching scheme* ('a branch is formed by assigning the next chosen value to the branching variable'), `KALIS_SMALLEST_DOMAIN` is the *variable selection strategy* ('choose the variable with the smallest number of values remaining in its domain'), and `KALIS_MIN_TO_MAX` the *value selection strategy* ('from smallest to largest value').

Since we are minimizing the number of time slots, enumeration starting with the smallest value seems to be a good idea. We therefore keep the default value selection criterion. However, we may try to change the variable selection heuristic: replacing `KALIS_SMALLEST_DOMAIN` by `KALIS_MAX_DEGREE` results in a reduction of the tree size and search time to less than half of its default size.

Here follows once more the complete model.

```
model "I-4 Scheduling exams (CP) - 4"
uses "kalis"

declarations
  NT: integer                ! Number of time slots
  EXAM: set of string        ! Set of exams
  TIME: set of integer       ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar ! Time slot for exam
  numslot: cpvar             ! Number of time slots used
end-declarations

initializations from 'Data/i4exam2.dat'
  INCOMP NT
end-initializations

finalize(EXAM)
TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
forall(e in EXAM) create(plan(e))

! Respect incompatibilities
forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e) plan(d) <> plan(e)

! Calculate number of time slots used
numslot = maximum(plan)

! Setting parameters of the enumeration
cp_set_branching(assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX))

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

! Solution printing
forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM)
    if (getsol(plan(e))=t) then write(e," "); end-if
  writeln
end-do

cp_show_stats

end-model
```

NB: In the model versions without optimization we may try to obtain a more evenly dis-

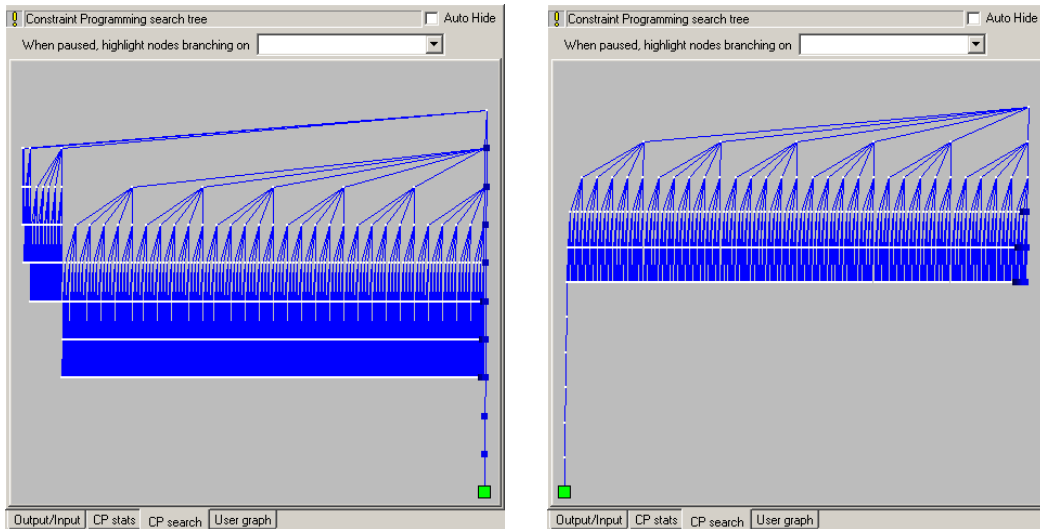


Figure 2.3: Search tree displays in IVE (left: default strategy, right: KALIS_MIN_DEGREE strategy for variable choice)

tributed schedule by choosing values randomly, that is, by using the value selection criterion KALIS_RANDOM_VALUE instead of KALIS_MIN_TO_MAX.

Further detail on the definition of branching strategies is given in Chapter 4.

2.3.2.1 IVE search tree display

With IVE, the difference between the two search trees becomes easily visible through the graphical display of the trees in the *CP search* pane (see Figure 2.3). Feasible solutions are represented by green squares, the optimal solution to an optimization problem is marked by a slightly larger square. As can be seen, with both strategies only a single solution is found. When hovering over the nodes in the search tree display detailed information including the node numbers and the name of the branching variables appears in pop-up boxes. It is also possible to highlight all nodes that branch on a given variable by indicating the variable's name in the selection box above the tree display. To obtain a more meaningful display in IVE, we may assign names to the decision variables using the procedure `setname` (see Section 2.1.2.3).

2.4 Continuous variables

All through this chapter we have worked with the decision variable type `cpvar` (discrete variables). A second variable type in Xpress-Kalis are continuous variables (type `cpfloatvar`). Such variables are used in a similar way to what we have seen above for discrete variables, for example:

```
setparam("DEFAULT_CONTINUOUS_LB", 0)
setparam("DEFAULT_CONTINUOUS_UB", 10)

declarations
  x,y: cpfloatvar
end-declarations

x >= y                                ! Define a constraint
                                     ! Retrieve information about continuous variables
writeln(getname(x), ":", getsol(x))
writeln(getlb(y), " ", getub(y))
```

A few differences in the use of the two decision variable types exist:

- Constraints involving `cpfloatvar` cannot be strict inequalities (that is, only the operators

`<=`, `>=`, and `=` may be used).

- Most global constraints (see Chapter 3) only apply to `cpvar`.
- Search strategies enumerating the values in a variable's domain can only be used with `cpvar` (see Chapter 4).
- Access functions for enumerating domain values such `getnext` are not applicable to `cpfloatvar`.

Chapter 3

Constraints

This chapter contains a collection of examples demonstrating the use of Xpress-Kalis for solving different types of (optimization) problems. The first section shows different ways of defining and posting constraints for simple linear constraints. The following sections each introduce a new constraint type. Since most examples use a combination of different constraints, the following list may help in finding examples of the use of a certain constraint type quickly.

- arithmetic: linear disequality: 2.1 (meeting.mos), 2.2 (exam*.mos), 3.7 (persplan.mos); linear equality/inequality: 3.5 (b4seq*_ka.mos), 3.6 (ilassign_ka.mos), 3.8 (b5paint*_ka.mos), 3.9 (j5tax_ka.mos), 4.4 (ilassign_ka.mos), 5.2 (blstadium_ka.mos); non-linear: 3.2
- all_different: 3.3 (sudoku_ka.mos), 3.5 (b4seq_ka.mos), 3.8 (b5paint*_ka.mos), 3.4 (freqasgn.mos), 3.7 (persplan.mos), 3.11 (eulerkn*.mos), 4.4 (ilassign_ka.mos)
- abs / distance: 3.4 (freqasgn.mos)
- cumulative: 5.4 (d4backup2_ka.mos)
- cycle: 3.10 (b5paint3_ka.mos)
- disjunctive: 3.5 (b4seq2_ka.mos)
- element: 3.5 (b4seq_ka.mos), 3.6 (a4sugar_ka.mos), 3.9 (j5tax_ka.mos), 3.8 (b5paint_ka.mos), 4.4 (ilassign_ka.mos), 2D: 3.8 (b5paint2_ka.mos)
- distribute / occurrence: 3.6 (a4sugar_ka.mos), 3.7 (persplan.mos), 3.9 (j5tax_ka.mos)
- maximum / minimum: 2.3 (exam3.mos, exam4.mos), 3.5 (b4seq2_ka.mos), 3.4 (freqasgn.mos), 4.4 (ilassign_ka.mos)
- implies / equiv: 3.8 (b5paint_ka.mos), 3.7 (persplan.mos), 3.9 (j5tax_ka.mos), 3.11 (eulerkn2.mos)
- generic binary: 3.11 (eulerkn.mos)

3.1 Constraint handling

In this section we shall work once more with the introductory problem of scheduling meetings from Section 2.1. This model only contains simple arithmetic constraints over discrete variables. However, all that is said here equally applies to all other constraint types of Xpress-Kalis, including constraint relations over continuous decision variables (that is, variables of the type `cpfloatvar`).

We now wish to state a few more constraints for this problem.

1. Meeting B must take place before day 3.
2. Meeting D cannot take place on day 2.
3. Meeting A must be scheduled on day 1.

3.1.1 Model formulation

The three additional constraints translate into simple (unary) linear constraints. We complete our model as follows:

$$\begin{aligned}
 \forall m \in \text{MEETINGS} : \text{plan}_m &\in \{1, 2, 3\} \\
 \text{plan}_B &\leq 2 \\
 \text{plan}_D &\neq 2 \\
 \text{plan}_A &= 1 \\
 \text{plan}_A &\neq \text{plan}_B \\
 \text{plan}_A &\neq \text{plan}_D \\
 \text{plan}_B &\neq \text{plan}_C \\
 \text{plan}_B &\neq \text{plan}_D
 \end{aligned}$$

3.1.2 Implementation

The Mosel implementation of the new constraints is quite straightforward.

```

model "Meeting (2)"
uses "kalis"

declarations
  MEETINGS = {'A','B','C','D'}           ! Set of meetings
  TIME = 1..3                             ! Set of time slots
  plan: array(MEETINGS) of cpvar         ! Time slot per meeting
end-declarations

forall(m in MEETINGS) do
  setdomain(plan(m), TIME)
  setname(plan(m), "plan"+m)
end-do
writeln("Original domains: ", plan)

plan('B') <= 2                           ! Meeting B before day 3
plan('D') <> 2                           ! Meeting D not on day 2
plan('A') = 1                           ! Meeting A on day 1
writeln("With constraints: ", plan)

! Respect incompatibilities
plan('A') <> plan('B')
plan('A') <> plan('D')
plan('B') <> plan('C')
plan('B') <> plan('D')

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

! Solution printing
forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model

```

3.1.3 Results

As the reader may have noticed, we have added printout of the variables plan_m at several

places in the model. The output generated by the execution of this model therefore is the following.

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
With constraints: [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

As can be seen from this output, immediately after stating the constraints, the domains of the concerned variables have been reduced. The constraints are *immediately and automatically posted* to the solver and their effects are propagated to the whole problem.

3.1.4 Naming constraints

Sometimes it may be necessary to access constraints later on, after their definition, in particular if they are to become part of logic relations or if they are to be branched on (typically the case for disjunctive constraints). To this aim Xpress-Kalis defines a new type, `cpctr`, that can be used to *declare constraints* giving them a name that can be used after their definition. We *define constraints* by assigning to them a constraint relation.

Naming constraints has the secondary effect that the constraints are *not automatically posted* to the solver. This needs to be done by writing the name of a constraint as a statement on its own (after its definition) as shown in the following model.

```
model "Meeting (3) "
  uses "kalis"

  declarations
    MEETINGS = {'A','B','C','D'}           ! Set of meetings
    TIME = 1..3                             ! Set of time slots
    plan: array(MEETINGS) of cpvar          ! Time slot per meeting
    Ctr: array(range) of cpctr
  end-declarations

  forall(m in MEETINGS) do
    setdomain(plan(m), TIME)
    setname(plan(m), "plan"+m)
  end-do
  writeln("Original domains: ", plan)

  Ctr(1) := plan('B') <= 2                  ! Meeting B before day 3
  Ctr(2) := plan('D') <> 2                  ! Meeting D not on day 2
  Ctr(3) := plan('A') = 1                  ! Meeting A on day 1
  writeln("After definition of constraints:\n ", plan)

  forall(i in 1..3) Ctr(i)
  writeln("After posting of constraints:\n ", plan)

  ! Respect incompatibilities
  plan('A') <> plan('B')
  plan('A') <> plan('D')
  plan('B') <> plan('C')
  plan('B') <> plan('D')

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

  ! Solution printing
  forall(m in MEETINGS) writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

From the output produced by this model, we can see that the mere definition of named con-

straints does not have any effect on the domains of the variables (the constraints are defined in Mosel but not yet sent to the Kalis solver). Only after stating the names of the constraints (that is, sending them to the solver) we obtain the same domain reductions as with the previous version of the model.

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
After definition of constraints:
  [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
After posting of constraints:
  [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

3.1.5 Explicit posting of constraints

In all previous examples, we have silently assumed that posting the constraints does not lead to a failure (infeasibility detected by the solver). In practice, this may not always be a reasonable assumption. Xpress-Kalis therefore defines the function `cp_post` that explicitly posts a constraint to the solver and returns the status of the constraint system after its addition (`true` for feasible and `false` for infeasible). This functionality may be of special interest for dealing with over-constrained problems to stop the addition of constraints once an infeasibility has been detected and to report back to the user which constraint has made the problem infeasible.

To use the explicit posting with `cp_post`, in the previous model we replace the line

```
forall(i in 1..3) Ctr(i)
```

with the following code.

```
forall(i in 1..3)
  if not cp_post(Ctr(i)) then
    writeln("Constraint ", i, " makes problem infeasible")
    exit(1)
  end-if
```

The return value of the every constraint posting is checked and the program is stopped if the addition of a constraint leads to an infeasibility.

The output produced by this model version is exactly the same as what we have seen in the previous section.

3.1.6 Explicit constraint propagation

The behavior of constraints can also be influenced in a different way. If we turn automated constraint propagation off,

```
setparam("AUTO_PROPAGATE", false)
```

then constraints posted to the solver are not propagated. In this case, constraint propagation will only be launched by a call to `cp_propagate` or by starting the enumeration (subroutines `cp_find_next_sol`, `cp_minimize`, *etc.*).

3.2 Arithmetic constraints

In the previous sections we have already seen several examples of linear constraints over finite domain variables. Linear constraints may be regarded as a special case of arithmetic constraints, that is, equations or inequality relations involving expressions over decision variables formed with the operators `+`, `-`, `/`, `*`, `^`, `sum`, `prod` and arithmetic functions like `abs` or `ln`. For

a complete list of arithmetic functions supported by the solver the reader is referred to the Xpress-Kalis reference manual.

Arithmetic constraints in Xpress-Kalis may be defined over finite domain variables (type `cpvar`), continuous variables (type `cpfloatvar`), or mixtures of both. Notice, however, that arithmetic constraints involving continuous variables cannot be defined as strict inequalities, that means, only the relational operators \geq , \leq , and $=$ may be used.

Here are a few examples of (nonlinear) arithmetic constraints that may be defined with Xpress-Kalis.

```
model "Nonlinear constraints"
uses "kalis"

setparam("DEFAULT_LB", 0)
setparam("DEFAULT_UB", 5)
setparam("DEFAULT_CONTINUOUS_LB", -10)
setparam("DEFAULT_CONTINUOUS_UB", 10)

declarations
  a,b,c: cpvar
  x,y,z: cpfloatvar
end-declarations

x = ln(y)
y = abs(z)
x*y <= z^2
z = -a/b
a*b*c^3 >= 150

while (cp_find_next_sol)
  writeln("a:", getsol(a), ", b:", getsol(b), ", c:", getsol(c),
    ", x:", getsol(x), ", y:", getsol(y), ", z:", getsol(z))
end-model
```

3.3 all_different: Sudoku

Sudoku puzzles, originating from Japan, have recently made their appearance in many western newspapers. The idea of these puzzles is to complete a given, partially filled 9×9 board with the numbers 1 to 9 in such a way that no line, column, or 3×3 subsquare contains a number more than once. The tables 3.1 and 3.2 show two instances of such puzzles. Whilst sometimes tricky to solve for a human, these puzzles lend themselves to solving by a CP approach.

Table 3.1: Sudoku ('The Times', 26 January, 2005)

	A	B	C	D	E	F	G	H	I
1					4	3		6	
2		6	5				7		
3	8			7					3
4		5				1	3		7
5	1	2						8	4
6	9		7	5				2	
7	4					5			9
8			9				4	5	
9		3		4	6				

3.3.1 Model formulation

As in the examples, we denote the columns of the board by the set $XS = \{A, B, \dots, I\}$ and the

Table 3.2: Sudoku ('The Guardian', 29 July, 2005)

	A	B	C	D	E	F	G	H	I
1	8					3			
2		5					4		
3	2				7			6	
4				1					5
5			3				9		
6	6					4			
7		7			2				3
8			4					1	
9				9					8

rows by $YS = \{1, 2, \dots, 9\}$. For every x in XS and y in YS we define a decision variable v_{xy} taking as its value the number at the position (x, y) .

The only constraints in this problem are

- (1) all numbers in a row must be different,
- (2) all numbers in a column must be different,
- (3) all numbers in a 3×3 subsquare must be different.

These constraints can be stated with Xpress-Kalis's `all_different` relation. This constraint ensures that all variables in the relation take different values.

```

 $\forall x \in XS, y \in YS : v_{xy} \in \{1, \dots, 9\}$ 
 $\forall x \in XS : \text{all\_different}(v_{x1}, \dots, v_{x9})$ 
 $\forall y \in YS : \text{all\_different}(v_{Ay}, \dots, v_{Iy})$ 
 $\text{all\_different}(v_{A1}, \dots, v_{C3})$ 
 $\text{all\_different}(v_{A4}, \dots, v_{C6})$ 
 $\text{all\_different}(v_{A7}, \dots, v_{C9})$ 
 $\text{all\_different}(v_{D1}, \dots, v_{F3})$ 
 $\text{all\_different}(v_{D4}, \dots, v_{F6})$ 
 $\text{all\_different}(v_{D7}, \dots, v_{F9})$ 
 $\text{all\_different}(v_{G1}, \dots, v_{I3})$ 
 $\text{all\_different}(v_{G4}, \dots, v_{I6})$ 
 $\text{all\_different}(v_{G7}, \dots, v_{I9})$ 

```

In addition, certain variables v_{xy} are fixed to the given values.

3.3.2 Implementation

The Mosel implementation for the Sudoku puzzle in Table 3.2 looks as follows.

```

model "sudoku (CP) "
uses "kalis"

forward procedure print_solution(numsol: integer)

setparam("default_lb", 1)
setparam("default_ub", 9)           ! Default variable bounds

declarations
  XS = {'A','B','C','D','E','F','G','H','I'} ! Columns
  YS = 1..9 ! Rows
  v: array(XS,YS) of cpvar ! Number assigned to cell (x,y)
end-declarations

```

```

! Data from "The Guardian", 29 July, 2005. http://www.guardian.co.uk/sudoku
v('A',1)=8; v('F',1)=3
v('B',2)=5; v('G',2)=4
v('A',3)=2; v('E',3)=7; v('H',3)=6
v('D',4)=1; v('I',4)=5
v('C',5)=3; v('G',5)=9
v('A',6)=6; v('F',6)=4
v('B',7)=7; v('E',7)=2; v('I',7)=3
v('C',8)=4; v('H',8)=1
v('D',9)=9; v('I',9)=8

! All-different values in rows
forall(y in YS) all_different(union(x in XS) {v(x,y)})

! All-different values in columns
forall(x in XS) all_different(union(y in YS) {v(x,y)})

! All-different values in 3x3 squares
forall(s in {{'A','B','C'},{'D','E','F'},{'G','H','I'}}, i in 0..2)
  all_different(union(x in s, y in {1+3*i,2+3*i,3+3*i}) {v(x,y)})

! Solve the problem
solct:= 0
while (cp_find_next_sol) do
  solct+=1
  print_solution(solct)
end-do

writeln("Number of solutions: ", solct)
writeln("Time spent in enumeration: ", getparam("COMPUTATION_TIME"), "sec")
writeln("Number of nodes: ", getparam("NODES"))

!*****
! Solution printing
procedure print_solution(numsol: integer)
  writeln(getparam("COMPUTATION_TIME"), "sec: Solution ", numsol)
  writeln("   A B C   D E F   G H I")
  forall(y in YS) do
    write(y, ": ")
    forall(x in XS)
      write(getsol(v(x,y)), if(x in {'C','F'}, " | ", " "))
    writeln
    if y mod 3 = 0 then
      writeln("   -----")
    end-if
  end-do
end-procedure

end-model

!*****

```


In this model, the call to `cp_find_next_sol` is embedded in a while loop to search all feasible solutions. At every loop execution the procedure `print_solution` is called to print out the solution found nicely formatted. Subroutines in Mosel may have declarations blocks for local declarations and they may take any number of arguments. Since, in our model, the call to the procedure occurs before its definition, we need to declare it before the first call using the keyword `forward`.

For selecting the information that is to be printed by the subroutine we use two different versions of Mosel's `if` statement: the inline `if` and `if-then` that includes a block of statements.

At the end of the model run we retrieve from the solver the run time measurement (parameter `COMPUTATION_TIME`) and the number of nodes explored by the search (parameter `NODES`).

To obtain a complete list of parameters defined by Xpress-Kalis type the command

```
mosel -c "exam -p kalis"
```

(for a listing of the complete functionality of Xpress-Kalis leave out the flag `-p`). With IVE, select *Modules* \gg *List available modules* or alternatively, click on the button .

3.3.3 Results

The model shown above generates the following output; this puzzle has only one solution, as is usually the case for Sudoku puzzles.

```
0.16sec: Solution 1
  A B C   D E F   G H I
1: 8 6 9 | 2 4 3 | 1 5 7
2: 3 5 7 | 6 1 9 | 4 8 2
3: 2 4 1 | 8 7 5 | 3 6 9
-----
4: 4 9 8 | 1 3 2 | 6 7 5
5: 7 1 3 | 5 8 6 | 9 2 4
6: 6 2 5 | 7 9 4 | 8 3 1
-----
7: 1 7 6 | 4 2 8 | 5 9 3
8: 9 8 4 | 3 5 7 | 2 1 6
9: 5 3 2 | 9 6 1 | 7 4 8
-----
Number of solutions: 1
Time spent in enumeration: 0.41sec
Number of nodes: 2712
```

The `all_different` relation takes an optional second argument that allows the user to specify the *propagation algorithm* to be used for evaluating the constraint. If we change from the default setting (`KALIS_FORWARD_CHECKING`) to the more aggressive strategy `KALIS_GEN_ARC_CONSISTENCY` by adding this choice as the second argument, for example,

```
forall(y in YS)
  all_different(union(x in XS) {v(x,y)}, KALIS_GEN_ARC_CONSISTENCY)
```

we observe that the number of nodes is reduced to a single node—the problem is solved by simply posting the constraints. Whereas the time spent in the search is down to zero, the constraint posting now takes 4-5 times longer (still just a fraction of a second) due to the larger computational overhead of the generalized arc consistency algorithm. Allover, the time for problem definition and solving is reduced to less than a tenth of the previous time.

As a general rule, the generalized arc consistency algorithm achieves stronger pruning (*i.e.*, it removes more values from the domains of the variables). However, due to the increase in computation time its use is not always justified. The reader is therefore encouraged to try both algorithm settings in his models.

3.4 abs and distance: Frequency assignment

The area of telecommunications, and in particular mobile telecommunications, gives rise to many different variants of frequency assignment problems.

We are given a network of cells (nodes) with requirements of discrete frequency bands. Each cell has a given demand for a number of frequencies (bands). Figure 3.1 shows the structure of the network. Nodes linked by an edge are considered as neighbors. They must not be assigned the same frequencies to avoid interference. Furthermore, if a cell uses several frequencies they must all be different by at least 2. The objective is to minimize the total number of frequencies used in the network.

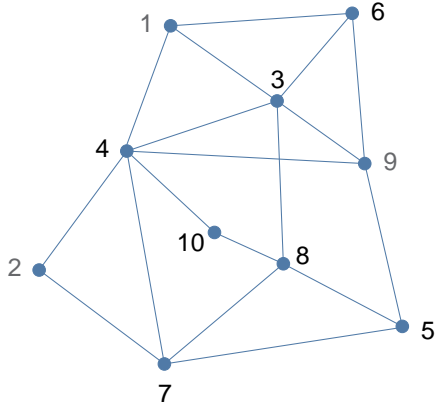


Figure 3.1: Telecommunications network

Table 3.3 lists the number of frequency demands for every cell.

Table 3.3: Frequency demands at nodes

Cell	1	2	3	4	5	6	7	8	9	10
Demand	4	5	2	3	2	4	3	4	3	2

3.4.1 Model formulation

Let $NODES$ be the set of all nodes in the network and DEM_n the demand of frequencies at node $n \in NODES$. The network is given as a set of edges $LINKS$. Furthermore, let $DEMANDS = \{1, 2, \dots, NUMDEM\}$ be the set of frequencies, numbered consecutively across all nodes where the upper bound $NUMDEM$ is given by the total number of demands. The auxiliary array $INDEX_n$ indicates the starting index in $DEMANDS$ for node n .

For representing the frequency assigned to every demand $d \in DEMANDS$ we introduce the variables use_d that take their values from the set $\{1, 2, \dots, NUMDEM\}$.

The two sets of constraints (different frequencies assigned to neighboring nodes and minimum distance between frequencies within a node) can then be modeled as follows.

$$\forall (n, m) \in LINKS : \text{all-different} \left(\bigcup_{d=INDEX_n}^{INDEX_n+DEM_n-1} use_d \cup \bigcup_{d=INDEX_m}^{INDEX_m+DEM_m-1} use_d \right)$$

$$\forall n \in NODES, \forall c < d \in INDEX_n, \dots, INDEX_n + DEM_n - 1 : |use_c - use_d| \geq 2$$

The objective function is to minimize to the number of frequencies used. We formulate this by minimizing the largest frequency number that occurs for the use_d variables:

$$\text{minimize } \text{maximum}_{d \in DEMANDS} (use_d)$$

3.4.2 Implementation

The edges forming the telecommunications network are modeled as a list `LINK`, where edge l is given as `(LINK(1, 1), LINK(1, 2))`.

For the implementation of the constraints on the values of frequencies assigned to the same node we have two equivalent choices with Kalis, namely using `abs` or `distance` constraints.

```
model "Frequency assignment"
uses "kalis"

forward procedure print_solution
```



```

declarations
  NODES = 1..10                                ! Range of nodes
  LINKS = 1..18                                ! Range of links between nodes
  DEM: array(NODES) of integer                 ! Demand of nodes
  LINK: array(LINKS,1..2) of integer           ! Neighboring nodes
  INDEX: array(NODES) of integer               ! Start index in 'use'
  NUMDEM: integer                             ! Upper bound on no. of freq.
end-declarations

DEM :: (1..10)[4, 5, 2, 3, 2, 4, 3, 4, 3, 2]
LINK:: (1..18,1..2)[1, 3, 1, 4, 1, 6,
                    2, 4, 2, 7,
                    3, 4, 3, 6, 3, 8, 3, 9,
                    4, 7, 4, 9, 4,10,
                    5, 7, 5, 8, 5, 9,
                    6, 9, 7, 8, 8,10]

NUMDEM:= sum(n in NODES) DEM(n)

! Correspondence of nodes and demand indices:
! use(d) d = 1, ..., DEM(1) correspond to the demands of node 1
!           d = DEM(1)+1, ..., DEM(1)+DEM(2) - " - node 2 etc.
INDEX(1):= 1
forall(n in NODES | n > 1) INDEX(n) := INDEX(n-1) + DEM(n-1)

declarations
  DEMANDS = 1..NUMDEM                          ! Range of frequency demands
  use: array(DEMANDS) of cpvar                 ! Frequency used for a demand
  numfreq: cpvar                              ! Number of frequencies used
  Strategy: array(range) of cpbranching
end-declarations

! Setting the domain of the decision variables
forall(d in DEMANDS) setdomain(use(d), 1, NUMDEM)

! All frequencies attached to a node must be different by at least 2
forall(n in NODES, c,d in INDEX(n)..INDEX(n)+DEM(n)-1 | c<d)
  distance(use(c), use(d)) >= 2
! abs(use(c) - use(d)) >= 2

! Neighboring nodes take all-different frequencies
forall(l in LINKS)
  all_different(
    union(d in INDEX(LINK(l,1))..INDEX(LINK(l,1))+DEM(LINK(l,1))-1) {use(d)} +
    union(d in INDEX(LINK(l,2))..INDEX(LINK(l,2))+DEM(LINK(l,2))-1) {use(d)},
    KALIS_GEN_ARC_CONSISTENCY)

! Objective function: minimize the number of frequencies used, that is,
! minimize the largest value assigned to 'use'
setname(numfreq, "NumFreq")
numfreq = maximum(use)

! Search strategy
Strategy(1):=assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX, use)
Strategy(2):=assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX, use)
cp_set_branching(Strategy(1))
setparam("MAX_COMPUTATION_TIME", 1)
cp_set_solution_callback("print_solution")

! Try to find solution(s) with strategy 1
if (cp_minimize(numfreq)) then
  cp_show_stats
  sol:=getsol(numfreq)
end-if

! Restart search with strategy 2
cp_reset_search
if sol>0 then
  numfreq <= sol-1
end-if
cp_set_branching(Strategy(2))
setparam("MAX_COMPUTATION_TIME", 1000)

if (cp_minimize(numfreq)) then

```

```

        cp_show_stats
    elif sol>0 then
        writeln("Optimality proven")
    else
        writeln("Problem has no solution")
    end-if

!*****
! **** Solution printout ****
procedure print_solution
    writeln("Number of frequencies: ", getsol(numfreq))
    writeln("Frequency assignment: ")
    forall(n in NODES) do
        write("Node ", n, ": ")
        forall(d in INDEX(n)..INDEX(n)+DEM(n)-1) write(getsol(use(d)), " ")
    end-do
end-procedure

end-model

```

With just the default search strategy this model finds a solution of value 11 but it runs for a long time without being able to prove optimality. When experimenting with different search strategies we have found that the strategy obtained by changing the variable selection criterion to `KALIS_MAX_DEGREE` is able to prove optimality easily once a good solution is known. This problem is therefore solved in two steps: First, we use the default strategy for finding a good solution. This search is stopped after one second by setting a *time limit*. The search is then *restarted* (previously, we needed to reset the search tree in the solver with `cp_reset_search`) with a second strategy and the bound on the objective value from the previous run.

To ease the experiments with different search strategies we have defined an array `Strategy` of type `cpbranching` that stores the different search strategy definitions.

Another new feature demonstrated by this implementation is the use of a *callback*, more precisely the *solution callback* of Xpress-Kalis. The solution callback is defined with a user subroutine that will be called by the solver whenever the search has found a solution. Its typical uses are logging or storing of intermediate solutions or performing some statistics. Our procedure `print_solution` simply prints out the solution that has been found.

Improving the problem formulation: we may observe that in our problem formulation all demand variables within a node and the constraints on these variables are entirely *symmetric*. In the absence of other constraints, we may reduce these symmetries by imposing an order on the *use* variables,

$$use_d + 1 \leq use_{d+1}$$

for demands d and $d + 1$ belonging to the same cell. Doing so, the problem is solved to optimality within less than 40 nodes using just the default strategy. We may take this a step further by writing

$$use_d + 2 \leq use_{d+1}$$

The addition of these constraints shortens the search by yet a few more nodes. They can even be used simply in replacement of the `abs` or `distance` constraints.

3.4.3 Results

An optimal solution to this problem uses 11 different frequencies. The model shown in the program listing prints out the following assignment of frequencies to nodes:

```

Node 1: 1 3 5 7
Node 2: 1 3 5 7 10
Node 3: 2 8
Node 4: 4 6 9
Node 5: 4 6
Node 6: 4 6 9 11
Node 7: 2 8 11

```

Node 8: 1 3 5 7
Node 9: 1 3 5
Node 10: 2 8

3.5 element: Sequencing jobs on a single machine

The problem described in this section is taken from Section 7.4 ‘Sequencing jobs on a bottleneck machine’ of the book ‘Applications of optimization with Xpress-MP’

The aim of this problem is to provide a model that may be used with different objective functions for scheduling operations on a single (bottleneck) machine. We shall see here how to minimize the total processing time, the average processing time, and the total tardiness.

A set of tasks (or jobs) is to be processed on a single machine. The execution of tasks is non-preemptive (that is, an operation may not be interrupted before its completion). For every task i its release date, duration, and due date are given in Table 3.4.

Table 3.4: Task time windows and durations

Job	1	2	3	4	5	6	7
Release date	2	5	4	0	0	8	9
Duration	5	6	8	4	2	4	2
Due date	10	21	15	10	5	15	22

What is the optimal value for each of the objectives: minimizing the total duration of the schedule (*makespan*), the mean processing time or the total tardiness (that is, the amount of time by which the completion of jobs exceeds their respective due dates)?

3.5.1 Model formulation 1

We are going to present two alternative model formulations. The first is closer to the Mathematical Programming formulation in ‘Applications of optimization with Xpress-MP’. The second uses disjunctive constraints and branching on these. In both model formulations we are going deal with the different objective functions in sequence, but the body of the models will remain the same.

To represent the sequence of jobs we introduce variables $rank_k$ ($k \in JOBS = \{1, \dots, NJ\}$) that take as value the number of the job in position (rank) k . Every job j takes a single position. This constraint can be represented by an *all-different* on the $rank_k$ variables:

$$all\text{-}different(rank_1, \dots, rank_{NJ})$$

The processing time dur_k for the job in position k is given by DUR_{rank_k} (where DUR_j denotes the duration given in the table in the previous section). Similarly, the release time rel_k is given by REL_{rank_k} (where REL_j denotes the given release date):

$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$

$$\forall k \in JOBS : rel_k = REL_{rank_k}$$

If $start_k$ is the start time of the job at position k , this value must be at least as great as the release date of the job assigned to this position. The completion time $comp_k$ of this job is the sum of its start time plus its duration:

$$\forall k \in JOBS : start_k \geq rel_k$$

$$\forall k \in JOBS : comp_k = start_k + dur_k$$

Another constraint is needed to specify that two jobs cannot be processed simultaneously. The job in position $k + 1$ must start after the job in position k has finished, hence the following constraints:

$$\forall k \in \{1, \dots, NJ - 1\} : start_{k+1} \geq comp_k$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule), or, equivalently, to minimize the completion time of the last job (job with rank NJ). The complete model is then given by the following (where $MAXTIME$ is a sufficiently large value, such as the sum of all release dates and all durations):

```

minimize compNJ
∀k ∈ JOBS : rankk ∈ JOBS
∀k ∈ JOBS : startk, compk ∈ {0, ..., MAXTIME}
∀k ∈ JOBS : durk ∈ {minj ∈ JOBS DURj, ..., maxj ∈ JOBS DURj}
∀k ∈ JOBS : relk ∈ {minj ∈ JOBS RELj, ..., maxj ∈ JOBS RELj}
all-different(rank1, ..., rankNJ)
∀k ∈ JOBS : durk = DURrankk
∀k ∈ JOBS : relk = RELrankk
∀k ∈ JOBS : startk ≥ relk
∀k ∈ JOBS : compk = startk + durk
∀k ∈ {1, ..., NJ - 1} : startk+1 ≥ startk + durk

```

Objective 2: For minimizing the average processing time, we introduce an additional variable *totComp* representing the sum of the completion times of all jobs. We add the following constraint to the problem to calculate *totComp*:

$$totComp = \sum_{k \in JOBS} comp_k$$

The new objective consists of minimizing the average processing time, or equivalently, minimizing the sum of the job completion times:

```

minimize totComp

```

Objective 3: If we now aim to minimize the total tardiness, we again introduce new variables—this time to measure the amount of time that jobs finish after their due date. We write *late_k* for the variable that corresponds to the tardiness of the job with rank k . Its value is the difference between the completion time of a job j and its due date DUE_j . If the job finishes before its due date, the value must be zero. We thus obtain the following constraints:

```

∀k ∈ JOBS : duek = DUErankk
∀k ∈ JOBS : latek ≥ compk - duek

```

For the formulation of the new objective function we introduce the variable *totLate* representing the total tardiness of all jobs. The objective now is to minimize the value of this variable:

```

minimize totLate
totLate = \sum_{k \in JOBS} late_k

```

3.5.2 Implementation of model 1

The Mosel implementation below solves the same problem three times, each time with a different objective, and prints the resulting solutions by calling the procedures `print_sol` and `print_sol3`.

```

model "B-4 Sequencing (CP)"
uses "kalis"

forward procedure print_sol
forward procedure print_sol3

```

```

declarations
  NJ = 7                                ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer           ! Release dates of jobs
  DUR: array(JOBS) of integer           ! Durations of jobs
  DUE: array(JOBS) of integer           ! Due dates of jobs

  rank: array(JOBS) of cpvar            ! Number of job at position k
  start: array(JOBS) of cpvar           ! Start time of job at position k
  dur: array(JOBS) of cpvar             ! Duration of job at position k
  comp: array(JOBS) of cpvar            ! Completion time of job at position k
  rel: array(JOBS) of cpvar             ! Release date of job at position k
end-declarations

initializations from 'Data/b4seq.dat'
  DUR REL DUE
end-initializations

MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)
MINDUR:= min(j in JOBS) DUR(j); MAXDUR:= max(j in JOBS) DUR(j)
MINREL:= min(j in JOBS) REL(j); MAXREL:= max(j in JOBS) REL(j)

forall(j in JOBS) do
  1 <= rank(j); rank(j) <= NJ
  0 <= start(j); start(j) <= MAXTIME
  MINDUR <= dur(j); dur(j) <= MAXDUR
  0 <= comp(j); comp(j) <= MAXTIME
  MINREL <= rel(j); rel(j) <= MAXREL
end-do

! One position per job
all_different(rank)

! Duration of job at position k
forall(k in JOBS) dur(k) = element(DUR, rank(k))

! Release date of job at position k
forall(k in JOBS) rel(k) = element(REL, rank(k))

! Sequence of jobs
forall(k in 1..NJ-1) start(k+1) >= start(k) + dur(k)

! Start times
forall(k in JOBS) start(k) >= rel(k)

! Completion times
forall(k in JOBS) comp(k) = start(k) + dur(k)

! Set the branching strategy
cp_set_branching(split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))

!*** Objective function 1: minimize latest completion time ***
if cp_minimize(comp(NJ)) then
  print_sol
end-if

!*** Objective function 2: minimize average completion time ***
declarations
  totComp: cpvar
end-declarations

totComp = sum(k in JOBS) comp(k)

if cp_minimize(totComp) then
  print_sol
end-if

!*** Objective function 3: minimize total tardiness ***
declarations
  late: array(JOBS) of cpvar           ! Lateness of job at position k

```

```

    due: array(JOBS) of cpvar          ! Due date of job at position k
    totLate: cpvar
end-declarations

MINDUE:= min(k in JOBS) DUE(k); MAXDUE:= max(k in JOBS) DUE(k)

forall(k in JOBS) do
    MINDUE <= due(k); due(k) <= MAXDUE
    0 <= late(k); late(k) <= MAXTIME
end-do

! Due date of job at position k
forall(k in JOBS) due(k) = element(DUE, rank(k))

! Late jobs: completion time exceeds the due date
forall(k in JOBS) late(k) >= comp(k) - due(k)

totLate = sum(k in JOBS) late(k)

if cp_minimize(totLate) then
    writeln("Tardiness: ", getsol(totLate))
    print_sol
    print_sol3
end-if

!-----

! Solution printing
procedure print_sol
    writeln("Completion time: ", getsol(comp(NJ)) ,
           " average: ", getsol(sum(k in JOBS) comp(k)))
    write("\t")
    forall(k in JOBS) write(strfmt(getsol(rank(k)),4))
    write("\nRel\t")
    forall(k in JOBS) write(strfmt(getsol(rel(k)),4))
    write("\nDur\t")
    forall(k in JOBS) write(strfmt(getsol(dur(k)),4))
    write("\nStart\t")
    forall(k in JOBS) write(strfmt(getsol(start(k)),4))
    write("\nEnd\t")
    forall(k in JOBS) write(strfmt(getsol(comp(k)),4))
    writeln
end-procedure

procedure print_sol3
    write("Due\t")
    forall(k in JOBS) write(strfmt(getsol(due(k)),4))
    write("\nLate\t")
    forall(k in JOBS) write(strfmt(getsol(late(k)),4))
    writeln
end-procedure

end-model

```

NB: The reader may have been wondering why we did not use the more obvious pair *start*–*end* for naming the variables in this example: *end* is a keyword of the Mosel language (see the list of reserved words in the [Mosel language reference manual](#)), which means that neither *end* nor *END* may be redefined by a Mosel program. It is possible though, to use versions combining lower and upper case letters, like *End*, but to prevent any possible confusion we do not recommend their use.

3.5.3 Results

The minimum makespan of the schedule is 31, the minimum sum of completion times is 103 (which gives an average of $103 / 7 = 14.71$). A schedule with this objective value is $5 \rightarrow 4 \rightarrow 1 \rightarrow 7 \rightarrow 6 \rightarrow 2 \rightarrow 3$. If we compare the completion times with the due dates we see that jobs 1, 2, 3, and 6 finish late (with a total tardiness of 21). The minimum tardiness is 18. A schedule with this tardiness is $5 \rightarrow 1 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 7 \rightarrow 3$ where jobs 4 and 7 finish one time unit late and job 3 is late by 16 time units, and it terminates at time 31 instead of being ready at its due

date, time 15. This schedule has an average completion time of 15.71.

3.5.4 Alternative formulation using disjunctions

Our second model formulation is possibly a more straightforward way of representing the problem. It introduces disjunctive constraints and branching on these.

Every job is now represented by its starting time, variables $start_j$ ($j \in JOBS = \{1, \dots, NJ\}$) that take their values in $\{REL_j, \dots, MAXTIME\}$ (where $MAXTIME$ is a sufficiently large value, such as the sum of all release dates and all durations, and REL_j the release date of job j). We state the disjunctions as a single *disjunctive* relation on the start times and durations of all jobs.

$$disjunctive([start_1, \dots, start_{NJ}], [DUR_1, \dots, DUR_{NJ}])$$

This constraint replaces the pair-wise disjunctions

$$start_i + DUR_i \leq start_j \vee start_j + DUR_j \leq start_i$$

for all pairs of jobs $i < j \in JOBS$.

The processing time dur_k for the job in position k is given by DUR_{rank_k} (where DUR_j denotes the duration given in the table in the previous section). Similarly, its release time rel_k is given by REL_{rank_k} (where REL_j denotes the given release date).

$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$

$$\forall k \in JOBS : rel_k = REL_{rank_k}$$

The completion time $comp_j$ of a job j is the sum of its start time plus its duration DUR_j .

$$\forall j \in JOBS : comp_j = start_j + DUR_j$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule) or, equivalently, to minimize the completion time *finish* of the last job. The complete model is then given by the following (where $MAXTIME$ is a sufficiently large value, such as the sum of all release dates and all durations):

minimize *finish*

$$finish = maximum_{j \in JOBS}(comp_j)$$

$$\forall j \in JOBS : comp_j \in \{0, \dots, MAXTIME\}$$

$$\forall j \in JOBS : start_j \in \{REL_j, \dots, MAXTIME\}$$

$$disjunctive([start_1, \dots, start_{NJ}], [DUR_1, \dots, DUR_{NJ}])$$

$$\forall j \in JOBS : comp_j = start_j + DUR_j$$

Objective 2: The formulation of the second objective (minimizing the average processing time or, equivalently, minimizing the sum of the job completion times) remains unchanged from the first model—we introduce an additional variable *totComp* representing the sum of the completion times of all jobs.

minimize *totComp*

$$totComp = \sum_{k \in JOBS} comp_k$$

Objective 3: To formulate the objective of minimizing the total tardiness, we introduce new variables *late_j* to measure the amount of time that a job finishes after its due date. The value of these variables corresponds to the difference between the completion time of a job j and its due date DUE_j . If the job finishes before its due date, the value must be zero. The objective

now is to minimize the sum of these tardiness variables:

$$\begin{aligned} &\text{minimize } totLate \\ &totLate = \sum_{j \in JOBS} late_j \\ &\forall j \in JOBS : late_j \in \{0, \dots, MAXTIME\} \\ &\forall j \in JOBS : late_j \geq comp_j - DUE_j \end{aligned}$$

3.5.5 Implementation of model 2

As with model 1, the Mosel implementation below solves the same problem three times, each time with a different objective, and prints the resulting solutions by calling the procedures `print_sol` and `print_sol3`.

```
model "B-4 Sequencing (CP)"
uses "kalis"

forward procedure print_sol
forward procedure print_sol3

declarations
  NJ = 7                                ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer           ! Release dates of jobs
  DUR: array(JOBS) of integer           ! Durations of jobs
  DUE: array(JOBS) of integer           ! Due dates of jobs
  DURS: array(set of cpvar) of integer  ! Dur.s indexed by start variables

  start: array(JOBS) of cpvar           ! Start time of jobs
  comp: array(JOBS) of cpvar            ! Completion time of jobs
  finish: cpvar                         ! Completion time of the entire schedule
  Disj: set of cpctr                    ! Disjunction constraints
  Strategy: array(range) of cpbranching ! Branching strategy
end-declarations

initializations from 'Data/b4seq.dat'
  DUR REL DUE
end-initializations

MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

forall(j in JOBS) do
  0 <= start(j); start(j) <= MAXTIME
  0 <= comp(j); comp(j) <= MAXTIME
end-do

! Disjunctions between jobs
forall(j in JOBS) DURS(start(j)):= DUR(j)
disjunctive(union(j in JOBS) {start(j)}, DURS, Disj, 1)

! Start times
forall(j in JOBS) start(j) >= REL(j)

! Completion times
forall(j in JOBS) comp(j) = start(j) + DUR(j)

!**** Objective function 1: minimize latest completion time ****
finish = maximum(comp)

Strategy(1):= settle_disjunction(Disj)
Strategy(2):= split_domain(KALIS_LARGEST_MAX, KALIS_MIN_TO_MAX)
cp_set_branching(Strategy)

if cp_minimize(finish) then
  print_sol
end-if

!**** Objective function 2: minimize average completion time ****
```



```

declarations
  totComp: cpvar
end-declarations

totComp = sum(k in JOBS) comp(k)

if cp_minimize(totComp) then
  print_sol
end-if

!**** Objective function 3: minimize total tardiness ****
declarations
  late: array(JOBS) of cpvar      ! Lateness of jobs
  totLate: cpvar
end-declarations

forall(k in JOBS) do
  0 <= late(k); late(k) <= MAXTIME
end-do

! Late jobs: completion time exceeds the due date
forall(j in JOBS) late(j) >= comp(j) - DUE(j)

totLate = sum(k in JOBS) late(k)
if cp_minimize(totLate) then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
end-if

!-----

! Solution printing
procedure print_sol
  writeln("Completion time: ", getsol(finish) ,
    " average: ", getsol(sum(j in JOBS) comp(j)))
  write("Rel\t")
  forall(j in JOBS) write(strfmt(REL(j),4))
  write("\nDur\t")
  forall(j in JOBS) write(strfmt(DUR(j),4))
  write("\nStart\t")
  forall(j in JOBS) write(strfmt(getsol(start(j)),4))
  write("\nEnd\t")
  forall(j in JOBS) write(strfmt(getsol(comp(j)),4))
  writeln
end-procedure

procedure print_sol3
  write("Due\t")
  forall(j in JOBS) write(strfmt(DUE(j),4))
  write("\nLate\t")
  forall(j in JOBS) write(strfmt(getsol(late(j)),4))
  writeln
end-procedure

end-model

```

This implementation introduces a new branching scheme, namely `settle_disjunction`. As opposed to the branching strategies we have seen so far this scheme defines a branching strategy over *constraints*, and not over variables. With this scheme a node is created by choosing a constraint from the given set and the branches from the node are obtained by adding one of the mutually exclusive constraints forming this disjunctive constraint to the constraint system.

NB: the `disjunctive` constraint of Xpress-Kalis establishes pair-wise inequalities between the processing times of tasks. However, the definition of disjunctive constraints is not restricted to this case: a disjunction may have more than two components, and involve constraints of any type, including other logic relations obtained by combining constraints with `and` or `or`.

3.6 occurrence: Sugar production

The problem description in this section is taken from Section 6.4 ‘Cane sugar production’ of the book ‘Applications of optimization with Xpress-MP’

The harvest of cane sugar in Australia is highly mechanized. The sugar cane is immediately transported to a sugarhouse in wagons that run on a network of small rail tracks. The sugar content of a wagonload depends on the field it has been harvested from and on the maturity of the sugar cane. Once harvested, the sugar content decreases rapidly through fermentation and the wagonload will entirely lose its value after a certain time. At this moment, eleven wagons loaded with the same quantity have arrived at the sugarhouse. They have been examined to find out the hourly loss and the remaining life span (in hours) of every wagon, these data are summarized in the following table.

Table 3.5: Properties of the lots of cane sugar

Lot	1	2	3	4	5	6	7	8	9	10	11
Loss (kg/h)	43	26	37	28	13	54	62	49	19	28	30
Life span (h)	8	8	2	8	4	8	8	8	8	8	8

Every lot may be processed by any of the three, fully equivalent production lines of the sugarhouse. The processing of a lot takes two hours. It must be finished at the latest at the end of the life span of the wagonload. The manager of the sugarhouse wishes to determine a production schedule for the currently available lots that minimizes the total loss of sugar.

3.6.1 Model formulation

Let $WAGONS = \{1, \dots, NW\}$ be the set of wagons, NL the number of production lines and DUR the duration of the production process for every lot. The hourly loss for every wagon w is given by $LOSS_w$ and its life span by $LIFE_w$. We observe that, in an optimal solution, the production lines need to work without any break—otherwise we could reduce the loss in sugar by advancing the start of the lot that follows the break. This means that the completion time of every lot is of the form $s \cdot DUR$, with $s > 0$ and is an integer. The maximum value of s is the number of time slots (of length DUR) that the sugarhouse will work, namely $NS = \text{ceil}(NW / NL)$, where ceil stands for ‘rounded to the next largest integer’. If NW / NL is an integer, every line will process exactly NS lots. Otherwise, some lines will process $NS - 1$ lots, but at least one line processes NS lots. In all cases, the length of the optimal schedule is $NS \cdot DUR$ hours. We call $SLOTS = \{1, \dots, NS\}$ the set of time slots.

Every lot needs to be assigned to a time slot. We define variables $process_w$ for the time slot assigned to wagon w and variables $loss_w$ for the loss incurred by this wagonload. Every time slot may take up to NL lots because there are NL parallel lines; therefore, we limit the number of occurrences of time slot values among the $process_w$ variables (this constraint relation is often called *cardinality constraint*):

$$s \in SLOTS : |process_w = s|_{w \in WAGONS} \leq NL$$

The loss of sugar per wagonload w and time slot s is $COST_{ws} = s \cdot DUR \cdot LOSS_w$. Let variables $loss_w$ denote the loss incurred by wagon load w :

$$\forall w \in WAGONS : loss_w = COST_{w, process_w}$$

The objective function (total loss of sugar) is then given as the sum of all losses:

$$\text{minimize } \sum_{w \in WAGONS} loss_w$$

3.6.2 Implementation

The following model is the Mosel implementation of this problem. It uses the function `ceil` to calculate the maximum number of time slots.

The constraints on the processing variables are expressed by `occurrence` relations and the losses are obtained via `element` constraints. The branching strategy uses the variable selection criterion `KALIS_SMALLEST_MAX`, that is, choosing the variable with the smallest upper bound.

```

model "A-4 Cane sugar production (CP)"
uses "kalis"

declarations
  NW = 11                                ! Number of wagon loads of sugar
  NL = 3                                ! Number of production lines
  WAGONS = 1..NW
  NS = ceil(NW/NL)
  SLOTS = 1..NS                          ! Time slots for production

  LOSS: array(WAGONS) of integer          ! Loss in kg/hour
  LIFE: array(WAGONS) of integer          ! Remaining time per lot (in hours)
  DUR: integer                            ! Duration of the production (in hours)
  COST: array(SLOTS) of integer           ! Cost per wagon

  loss: array(WAGONS) of cpvar            ! Loss per wagon
  process: array(WAGONS) of cpvar         ! Time slots for wagon loads

  totalLoss: cpvar                        ! Objective variable
end-declarations

initializations from 'Data/a4sugar.dat'
  LOSS LIFE DUR
end-initializations

forall(w in WAGONS) setdomain(process(w), 1, NS)

! Wagon loads per time slot
forall(s in SLOTS) occurrence(s, process) <= NL

! Limit on raw product life
forall(w in WAGONS) process(w) <= floor(LIFE(w)/DUR)

! Objective function: total loss
forall(w in WAGONS) do
  forall(s in SLOTS) COST(s) := s*DUR*LOSS(w)
  loss(w) = element(COST, process(w))
end-do
totalLoss = sum(w in WAGONS) loss(w)

cp_set_branching(assign_var(KALIS_SMALLEST_MAX, KALIS_MIN_TO_MAX, process))

! Solve the problem
if not (cp_minimize(totalLoss)) then
  writeln("No solution found")
  exit(0)
end-if

! Solution printing
writeln("Total loss: ", getsol(totalLoss))
forall(s in SLOTS) do
  write("Slot ", s, ": ")
  forall(w in WAGONS)
    if(getsol(process(w))=s) then
      write("wagon ", strfmt(w,2), strfmt(" (" + s*DUR*LOSS(w) + ") ", 8))
    end-if
  writeln
end-do

end-model

```

An alternative formulation of the constraints on the processing variables is to replace them by a single `distribute` relation, indicating for every time slot the minimum and maximum number ($MINUSE_s = 0$ and $MAXUSE_s = NL$) of production lines that may be used.

```

forall(s in SLOTS) MAXUSE(s) := NL
distribute(process, SLOTS, MINUSE, MAXUSE)

```

Yet another formulation of this problem is possible with Xpress-Kalis, namely interpreting it as a cumulative scheduling problem (see Section 5.4), where the wagon loads are represented by tasks of unit duration, scheduled on a discrete resource with a capacity corresponding to the number of production lines.

3.6.3 Results

We obtain a total loss of 1620 kg of sugar. The corresponding schedule of lots is shown in the following Table 3.6 (there are several equivalent solutions).

Table 3.6: Optimal schedule for the cane sugar lots

Slot 1	Slot 2	Slot 3	Slot 4
lot 3 (74 kg)	lot 1 (172 kg)	lot 4 (168 kg)	lot 2 (208 kg)
lot 6 (108 kg)	lot 5 (52 kg)	lot 9 (114 kg)	lot 10 (224 kg)
lot 7 (124 kg)	lot 8 (196 kg)	lot 11 (180 kg)	

3.7 distribute: Personnel planning

The director of a movie theater wishes to establish a plan with the working locations for his personnel. The theater has eight employees: David, Andrew, Leslie, Jason, Oliver, Michael, Jane, and Marilyn. The ticket office needs to be staffed with three persons, theater entrances one and two require two persons each, and one person needs to be put in charge of the cloakroom. Due to different qualifications and respecting individual likes and dislikes, the following constraints need to be taken into account:

1. Leslie must be at the second entrance of the theater.
2. Michael must be at the first entrance of the theater.
3. David, Michael and Jason cannot work with each other.
4. If Oliver is selling tickets, Marylin must be with him.

3.7.1 Model formulation

Let $PERS$ be the set of personnel and $LOC = \{1, \dots, 4\}$ the set of working locations where 1 stands for the ticket office, 2 and 3 for entrances 1 and 2 respectively, and 4 for the cloakroom.

We introduce decision variables $place_p$ for the working location assigned to person p . The four individual constraints on working locations can then be stated as follows:

$$\begin{aligned}
 place_{Leslie} &= 3 \\
 place_{Michael} &= 2 \\
 all\text{-}different(place_{David}, place_{Michael}, place_{Jason}) \\
 place_{Oliver} = 1 &\Rightarrow place_{Marylin} = 1
 \end{aligned}$$

We also have to meet the staffing requirement of every working location:

$$\forall l \in LOC : |place_p = l|_{p \in PERS} = REQ_l$$

3.7.2 Implementation

For the implementation of the staffing requirements we may choose among two different constraints of Xpress-Kalis, namely `occurrence` constraints (one per working location) or a

distribute constraint (also known as *global cardinality constraint*) over all working locations. The following model shows both options. As opposed to the previous example (Section 3.6.2 we now have equality constraints. We therefore use the three-argument version of distribute (instead of the version with four arguments where the last two are the lower and upper bounds respectively).

For an attractive display, the model also introduces a few auxiliary data structures with the names of the working locations and the corresponding index values in the set LOC.

```

model "Personnel Planning (CP)"
  uses "kalis"

  forward procedure print_solution

  declarations
    PERS = {"David", "Andrew", "Leslie", "Jason", "Oliver", "Michael",
           "Jane", "Marilyn"}           ! Set of personnel
    LOC = 1..4                           ! Set of locations
    LOCNAMES = {"Ticketoffice", "Theater1", "Theater2",
               "Cloakroom"}              ! Names of locations
    LOCNUM: array(LOCNAMES) of integer   ! Numbers assoc. with loc.s
    REQ: array(LOC) of integer           ! No. of pers. req. per loc.

    place: array(PERS) of cpvar          ! Workplace assigned to each person
  end-declarations

  ! Initialize data
  LOCNUM("Ticketoffice") := 1; LOCNUM("Theater1") := 2
  LOCNUM("Theater2") := 3; LOCNUM("Cloakroom") := 4
  REQ:: (1..4) [3, 2, 2, 1]

  ! Each variable has a lower bound of 1 (Ticketoffice) and an upper bound
  ! of 4 (Cloakroom)
  forall(p in PERS) do
    setname(place(p), "workplace["+p+"]")
    setdomain(place(p), LOC)
  end-do

  ! "Leslie must be at the second entrance of the theater"
  place("Leslie") = LOCNUM("Theater2")

  ! "Michael must be at the first entrance of the theater"
  place("Michael") = LOCNUM("Theater1")

  ! "David, Michael and Jason cannot work with each other"
  all_different({place("David"), place("Michael"), place("Jason")})

  ! "If Oliver is selling tickets, Marilyn must be with him"
  implies(place("Oliver")=LOCNUM("Ticketoffice"),
         place("Marilyn")=LOCNUM("Ticketoffice"))

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

  ! Formulation of resource constraints using global cardinality constraint
  distribute(place, LOC, REQ)

  ! Setting parameters of the enumeration
  cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, place))

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

  ! Solution output
  nbSolutions:= 1
  print_solution

  ! Search for other solutions
  while (cp_find_next_sol) do

```

```

        nbSolutions += 1
        print_solution
    end-do

! **** Solution printout ****
procedure print_solution
    declarations
        LOCIDX: array(LOC) of string
    end-declarations
    forall(l in LOCNUMS) LOCIDX(LOCNUM(l)):=1

    writeln("\nSolution number ", nbSolutions)
    forall(p in PERS)
        writeln(" Working place of ", p, ": ", LOCIDX(getsol(place(p))))
    end-procedure

end-model

```

Since we merely wish to find a feasible assignment of working locations, this model first tests whether a feasible solution exists. If this is the case, it also enumerates all other feasible solutions. Each time a solution is found it is printed out by call to the procedure `print_solution`.

3.7.3 Results

This problem has 38 feasible solutions. The graphical representation with IVE of a feasible assignment is shown in Figure 3.2.

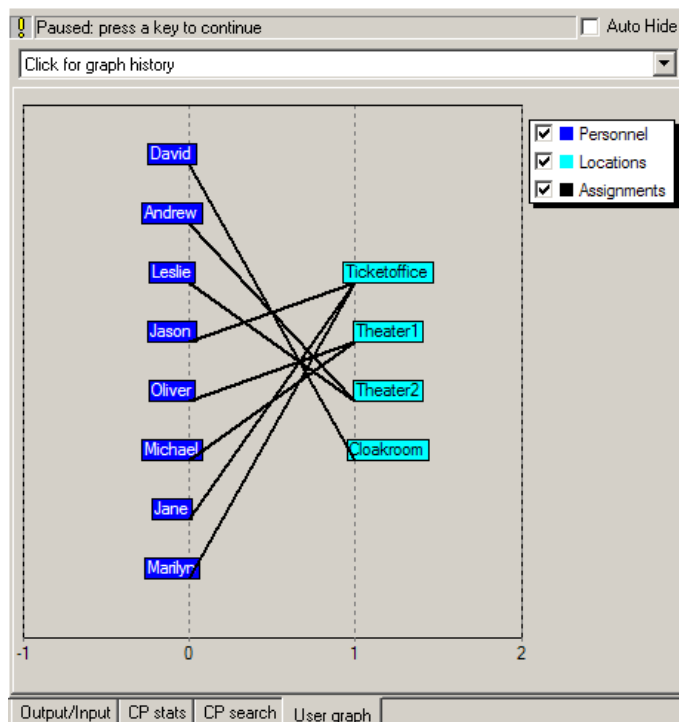


Figure 3.2: Personnel schedule representation in IVE

The following Mosel code was used for generating the graphic (see the documentation of module `mmive` in the [Mosel language reference manual](#) for further explanation).

```

procedure draw_solution
    IVErase
    PersGraph:= IVEaddplot("Personnel", IVE_BLUE)
    LocGraph:= IVEaddplot("Locations", IVE_CYAN)
    AsgnGraph:= IVEaddplot("Assignments", IVE_BLACK)

```

```

forall (d in LOCNUMS)
  IVEdrawlabel(LocGraph, 1.2, LOCNUM(d)-getsize(LOC)/2, d)

idx:= 1
forall(p in PERS) do
  IVEdrawline(AsgnGraph, 0, idx-getsize(PERS)/2,
              1, getsol(place(p))-getsize(LOC)/2)
  IVEdrawlabel(PersGraph, -0.1, idx-getsize(PERS)/2, p)
  idx:= idx + 1
end-do

IVEzoom(-1, getsize(PERS)/2+1, 2, -getsize(PERS)/2)
IVEpause("press a key to continue")
end-procedure

```

3.8 implies: Paint production

The problem description in this section is taken from Section 7.5 ‘Paint production’ of the book ‘Applications of optimization with Xpress-MP’

As a part of its weekly production a paint company produces five batches of paints, always the same, for some big clients who have a stable demand. Every paint batch is produced in a single production process, all in the same blender that needs to be cleaned between every two batches. The durations of blending paint batches 1 to 5 are respectively 40, 35, 45, 32, and 50 minutes. The cleaning times depend on the colors and the paint types. For example, a long cleaning period is required if an oil-based paint is produced after a water-based paint, or to produce white paint after a dark color. The times are given in minutes in the following Table 3.7 *CLEAN* where $CLEAN_{ij}$ denotes the cleaning time between batch i and batch j .

Table 3.7: Matrix of cleaning times

	1	2	3	4	5
1	0	11	7	13	11
2	5	0	13	15	15
3	13	15	0	23	11
4	9	13	5	0	3
5	3	7	7	7	0

Since the company also has other activities, it wishes to deal with this weekly production in the shortest possible time (blending and cleaning). Which is the corresponding order of paint batches? The order will be applied every week, so the cleaning time between the last batch of one week and the first of the following week needs to be counted for the total duration of cleaning.

3.8.1 Formulation of model 1

As for the problem in Section 3.5 we are going to present two alternative model formulations. The first one is closer to the Mathematical Programming formulation in ‘Applications of optimization with Xpress-MP’, the second uses a two-dimensional element constraint.

Let $JOBS = \{1, \dots, NJ\}$ be the set of batches to produce, DUR_j the processing time for batch j , and $CLEAN_{ij}$ the cleaning time between the consecutive batches i and j . We introduce decision variables $succ_j$ taking their values in $JOBS$, to indicate the successor of every job, and variables $clean_j$ for the duration of the cleaning after every job. The cleaning time after every job is obtained by indexing $CLEAN_{ij}$ with the value of $succ_j$. We thus have the following problem formulation.

$$\text{minimize } \sum_{j \in JOBS} (DUR_j + clean_j)$$

$$\forall j \in JOBS : succ_j \in JOBS \setminus \{j\}$$

$$\forall j \in JOBS : clean_j = CLEAN_{j, succ_j}$$

$$all\text{-}different\left(\bigcup_{j \in JOBS} succ_j\right)$$

The objective function sums up the processing and cleaning times of all batches. The last (all-different) constraint guarantees that every batch occurs exactly once in the production sequence.

Unfortunately, this model does not guarantee that the solution forms a single cycle. Solving it indeed results in a total duration of 239 with an invalid solution that contains two sub-cycles $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$ and $4 \rightarrow 5 \rightarrow 4$. A first possibility is to add a disjunction excluding this solution to our model and re-solve it iteratively until we reach a solution without sub-cycles.

$$\forall succ_1 \neq 3 \vee succ_3 \neq 2 \vee succ_2 \neq 1 \vee succ_1 \neq 5 \vee succ_5 \neq 4$$

However, this procedure is likely to become impractical with larger data sets since it may potentially introduce an extremely large number of disjunctions. We therefore choose a different, a-priori formulation of the sub-cycle elimination constraints with a variable y_j per batch and $NJ \cdot (NJ - 1)$ implication constraints.

$$\begin{aligned} \forall j \in JOBS : rank_j &\in \{1, \dots, NJ\} \\ \forall i \in JOBS, \forall j = 2, \dots, NJ, i \neq j : succ_i = j &\Rightarrow y_j = y_i + 1 \end{aligned}$$

The variables y_j correspond to the position of job j in the production cycle. With these constraints, job 1 always takes the first position.

3.8.2 Implementation of model 1

The Mosel implementation of the model formulated in the previous section is quite straightforward. The sub-cycle elimination constraints are implemented as logic relations with `implies` (a stronger formulation of these constraints is obtained by replacing the implications by equivalences, using `equiv`).

```
model "B-5 Paint production (CP)"
uses "kalis"

declarations
  NJ = 5                                ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer           ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer    ! Cleaning times between jobs
  CB: array(JOBS) of integer            ! Cleaning times after a batch

  succ: array(JOBS) of cpvar            ! Successor of a batch
  clean: array(JOBS) of cpvar           ! Cleaning time after batches
  y: array(JOBS) of cpvar               ! Variables for excluding subtours
  cycleTime: cpvar                     ! Objective variable
end-declarations

initializations from 'Data/b5paint.dat'
  DUR CLEAN
end-initializations

forall(j in JOBS) do
  1 <= succ(j); succ(j) <= NJ; succ(j) <> j
  1 <= y(j); y(j) <= NJ
end-do

! Cleaning time after every batch
forall(j in JOBS) do
  forall(i in JOBS) CB(i) := CLEAN(j,i)
  clean(j) = element(CB, succ(j))
end-do
```



```

! Objective: minimize the duration of a production cycle
cycleTime = sum(j in JOBS) (DUR(j)+clean(j))

! One successor and one predecessor per batch
all_different(succ)

! Exclude subtours
forall(i in JOBS, j in 2..NJ | i<>j)
  implies(succ(i) = j, y(j) = y(i) + 1)

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

! Solution printing
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:\nBatch Duration Cleaning")
first:=1
repeat
  writeln(" ", first, sprintf(DUR(first),8), sprintf(getsol(clean(first)),9))
  first:=getsol(succ(first))
until (first=1)

end-model

```

3.8.3 Formulation of model 2

We may choose to implement the paint production problem using rank variables similarly to the sequencing model in Section 3.5.1.

As before, let $JOBS = \{1, \dots, NJ\}$ be the set of batches to produce, DUR_j the processing time for batch j , and $CLEAN_{ij}$ the cleaning time between the consecutive batches i and j . We introduce decision variables $rank_k$ taking their values in $JOBS$, for the number of the job in position k . Variables $clean_k$ ($k \in JOBS$) now denote the duration of the k^{th} cleaning time. This duration is obtained by indexing $CLEAN_{ij}$ with the values of two consecutive $rank_k$ variables. We thus have the following problem formulation.

$$\begin{aligned}
& \text{minimize } \sum_{j \in JOBS} DUR_j + \sum_{k \in JOBS} clean_k \\
& \forall k \in JOBS : rank_k \in JOBS \\
& \forall k \in \{1, \dots, NJ - 1\} : clean_k = CLEAN_{rank_k, rank_{k+1}} \\
& clean_{NJ} = CLEAN_{rank_{NJ}, rank_1} \\
& all\text{-}different \left(\bigcup_{k \in JOBS} rank_k \right)
\end{aligned}$$

As in model 1, the objective function sums up the processing and cleaning times of all batches. Although not strictly necessary from the mathematical point of view, we use different sum indices for durations and cleaning times to show the difference between summing over jobs or job positions. We now have an all-different constraint over the rank variables to guarantee that every batch occurs exactly once in the production sequence.

3.8.4 Implementation of model 2

The implementation of the second model uses the 2-dimensional version of the `element` constraint in Xpress-Kalis.

```

model "B-5 Paint production (CP)"
  uses "kalis"

  declarations

```

```

NJ = 5                                ! Number of paint batches (=jobs)
JOBS=1..NJ

DUR: array(JOBS) of integer           ! Durations of jobs
CLEAN: array(JOBS,JOBS) of integer    ! Cleaning times between jobs

rank: array(JOBS) of cpvar            ! Number of job in position k
clean: array(JOBS) of cpvar           ! Cleaning time after batches
cycleTime: cpvar                      ! Objective variable
end-declarations

initializations from 'Data/b5paint.dat'
DUR CLEAN
end-initializations

forall(k in JOBS) setdomain(rank(k), JOBS)

! Cleaning time after every batch
forall(k in JOBS)
    element(CLEAN, rank(k), if(k<NJ, rank(k+1), rank(1))) = clean(k)

! Objective: minimize the duration of a production cycle
cycleTime = sum(j in JOBS) DUR(j) + sum(k in JOBS) clean(k)

! One position for every job
all_different(rank)

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

! Solution printing
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:\nBatch Duration Cleaning")
forall(k in JOBS)
    writeln("  ", getsol(rank(k)), strfmt(DUR(getsol(rank(k))),8),
            strfmt(getsol(clean(k)),9))

end-model

```

3.8.5 Results

The minimum cycle time for this problem is 243 minutes which is achieved with the following sequence of batches: $1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 1$. This time includes 202 minutes of (incompressible) processing time and 41 minutes of cleaning.

When comparing the problem statistics produced by Xpress-Kalis for this problem we see that the second model is a weaker formulation resulting in a considerably longer enumeration (using the default strategies).

3.9 equiv: Location of income tax offices

The example description in the following sections is taken from Section 15.5 'Location of income tax offices' of the book '[Applications of optimization with Xpress-MP](#)'.

The income tax administration is planning to restructure the network of income tax offices in a region. The number of inhabitants of every city and the distances between each pair of cities are known (Table 3.8). The income tax administration has determined that offices should be established in three cities to provide sufficient coverage. Where should these offices be located to minimize the average distance per inhabitant to the closest income tax office?

Table 3.8: Distance matrix and population of cities

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	15	37	55	24	60	18	33	48	40	58	67
2	15	0	22	40	38	52	33	48	42	55	61	61
3	37	22	0	18	16	30	43	28	20	58	39	39
4	55	40	18	0	34	12	61	46	24	62	43	34
5	24	38	16	34	0	36	27	12	24	49	37	43
6	60	52	30	12	36	0	57	42	12	50	31	22
7	18	33	43	61	27	57	0	15	45	22	40	61
8	33	48	28	46	12	42	15	0	30	37	25	46
9	48	42	20	24	12	45	30	0	38	19	19	19
10	40	55	58	62	49	50	22	37	38	0	19	40
11	58	61	39	43	37	31	40	25	19	19	0	21
12	67	61	39	34	43	22	61	46	19	40	21	0
Pop. (in 1000)	15	10	12	18	5	24	11	16	13	22	19	20

3.9.1 Model formulation

Let $CITIES$ be the set of cities. For the formulation of the problem, two groups of decision variables are necessary: a variable $build_c$ that is one if and only if a tax office is established in city c , and a variable $depend_c$ that takes the number of the office on which city c depends. For the formulation of the constraints, we further introduce two sets of auxiliary variables: $depdist_c$, the distance from city c to the office indicated by $depend_c$, and $numdep_c$, the number of cities depending on an office location.

The following relations are required to link the $build_c$ with the $depend_c$ variables:

- (1) $numdep_c$ counts the number of occurrences of office location c among the variables $depend_c$.
- (2) $numdep_c \geq 1$ if and only if the office in c is built (as a consequence, if the office in c is not built, then we must have $numdep_c = 0$).

$$\forall c \in CITIES : numdep_c = |depend_d = c|_{d \in CITIES}$$

$$\forall c \in CITIES : numdep_c \geq 1 \Leftrightarrow build_c = 1$$

Since the number of offices built is limited by the given bound $NUMLOC$

$$\sum_{c \in CITIES} build_c \leq NUMLOC$$

it would actually be sufficient to formulate the second relation between the $build_c$ and $depend_c$ variables as the implication 'If $numdep_c \geq 1$ then the office in c must be built, and inversely, if the office in c is not built, then we must have $numdep_c = 0$ '.

The objective function to be minimized is the total distance weighted by the number of inhabitants of the cities. We need to divide the resulting value by the total population of the region to obtain the average distance per inhabitant to the closest income tax office. The distance $depdist_c$ from city c to the closest tax office location is obtained by a discrete function, namely the row c of the distance matrix $DIST_{cd}$ indexed by the value of $depend_c$:

$$depdist_c = DIST_{c, depend_c}$$

We now obtain the following CP model:

$$\begin{aligned} & \text{minimize } \sum_{c \in CITIES} POP_c \cdot dist_c \\ & \forall c \in CITIES : build_c \in \{0, 1\}, \quad depend_c \in CITIES, \\ & numdep_c \in CITIES \cup \{0\}, \quad depdist_c \in \{\min_{d \in CITIES} DIST_{c,d}, \dots, \max_{d \in CITIES} DIST_{c,d}\} \\ & \forall c \in CITIES : depdist_c = DIST_{c, depend_c} \end{aligned}$$

$$\sum_{c \in \text{CITIES}} \text{build}_c \leq \text{NUMLOC}$$

$$\forall c \in \text{CITIES} : \text{numdep}_c = |\text{depend}_d = c|_{d \in \text{CITIES}}$$

$$\forall c \in \text{CITIES} : \text{numdep}_c \geq 1 \Leftrightarrow \text{build}_c = 1$$

3.9.2 Implementation

To solve this problem, we define a branching strategy with two parts, one for the build_c variables and a second strategy for the depdist_c variables. The latter are enumerated using the `split_domain` branching scheme that divides the domain of the branching variable into several disjoint subsets (instead of assigning a value to the variable). We now pass an array of type `cpbranching` as the argument to procedure `cp_set_branching`. The different strategies will be applied in their order in this array. Since our enumeration strategy does not explicitly include all decision variables of the problem, Xpress-Kalis will enumerate these using the default strategy if any unassigned variables remain after the application of our search strategy.

```

model "J-5 Tax office location (CP)"
uses "kalis"

forward procedure calculate_dist

setparam("DEFAULT_LB", 0)

declarations
  NC = 12
  CITIES = 1..NC                                ! Set of cities

  DIST: array(CITIES,CITIES) of integer ! Distance matrix
  POP: array(CITIES) of integer          ! Population of cities
  LEN: dynamic array(CITIES,CITIES) of integer ! Road lengths
  NUMLOC: integer                        ! Desired number of tax offices
  D: array(CITIES) of integer            ! Auxiliary array used in constr. def.

  build: array(CITIES) of cpvar          ! 1 if office in city, 0 otherwise
  depend: array(CITIES) of cpvar         ! Office on which city depends
  depdist: array(CITIES) of cpvar        ! Distance to tax office
  numdep: array(CITIES) of cpvar         ! Number of depending cities per off.
  totDist: cpvar                         ! Objective function variable
  Strategy: array(1..2) of cpbranching ! Branching strategy
end-declarations

initializations from 'Data/j5tax.dat'
  LEN POP NUMLOC
end-initializations

! Calculate the distance matrix
calculate_dist

forall(c in CITIES) do
  build(c) <= 1
  1 <= depend(c); depend(c) <= NC
  min(d in CITIES) DIST(c,d) <= depdist(c)
  depdist(c) <= max(d in CITIES) DIST(c,d)
  numdep(c) <= NC
end-do

! Distance from cities to tax offices
forall(c in CITIES) do
  forall(d in CITIES) D(d):=DIST(c,d)
  element(D, depend(c)) = depdist(c)
end-do

! Number of cities depending on every office
forall(c in CITIES) occurrence(c, depend) = numdep(c)

! Relations between dependencies and offices built
forall(c in CITIES) equiv( build(c) = 1, numdep(c) >= 1 )

! Limit total number of offices

```

```

sum(c in CITIES) build(c) <= NUMLOC

! Branching strategy
Strategy(1) := assign_and_forbid(KALIS_MAX_DEGREE, KALIS_MAX_TO_MIN, build)
Strategy(2) := split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX,
                           depdist, true, 5)
cp_set_branching(Strategy)

! Objective: weighted total distance
totDist = sum(c in CITIES) POP(c)*depdist(c)

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

! Solution printing
writeln("Total weighted distance: ", getsol(totDist),
        " (average per inhabitant: ",
        getsol(totDist)/sum(c in CITIES) POP(c), ")")
forall(c in CITIES) if(getsol(build(c))>0) then
  write("Office in ", c, ": ")
  forall(d in CITIES) write(if(getsol(depend(d))=c, " "+d, ""))
  writeln
end-if

!-----

! Calculate the distance matrix using Floyd-Warshall algorithm
procedure calculate_dist
! Initialize all distance labels with a sufficiently large value
BIGM:=sum(c,d in CITIES | exists(LEN(c,d))) LEN(c,d)
forall(c,d in CITIES) DIST(c,d):=BIGM

forall(c in CITIES) DIST(c,c):=0      ! Set values on the diagonal to 0

! Length of existing road connections
forall(c,d in CITIES | exists(LEN(c,d))) do
  DIST(c,d):=LEN(c,d)
  DIST(d,c):=LEN(c,d)
end-do

! Update shortest distance for every node triple
forall(b,c,d in CITIES | c<d )
  if DIST(c,d) > DIST(c,b)+DIST(b,d) then
    DIST(c,d) := DIST(c,b)+DIST(b,d)
    DIST(d,c) := DIST(c,b)+DIST(b,d)
  end-if
end-procedure

end-model

```

This implementation contains another example of the use of a subroutine in Mosel: the calculation of the distance data is carried out in the procedure `calculate_dist`. We thus use a subroutine to structure our model, removing secondary tasks from the main model formulation.

3.9.3 Results

The optimal solution to this problem has a total weighted distance of 2438. Since the region has a total of 185,000 inhabitants, the average distance per inhabitant is $2438/185 \approx 13.178$ km. The three offices are established at nodes 1, 6, and 11. The first serves cities 1, 2, 5, 7, the office in node 6 cities 3, 4, 6, 9, and the office in node 11 cities 8, 10, 11, 12.

3.10 `cyc1e`: Paint production

In this section we work once more with the paint production problem from Section 3.8. The

objective of this problem is to determine a production cycle of minimal length for a given set of jobs with sequence-dependent cleaning times between every pair of jobs.

3.10.1 Model formulation

The problem formulation in Section 3.8 uses 'all-different' constraints to ensure that every job occurs once only, calculates the duration of cleaning times with 'element' constraints, and introduces auxiliary variables and constraints to prevent subcycles in the production sequence. All these constraints can be expressed by a single constraint relation, the 'cycle' constraint.

Let $JOBS = \{1, \dots, NJ\}$ be the set of batches to produce, DUR_j the processing time for batch j , and $CLEAN_{ij}$ the cleaning time between the consecutive batches i and j . As before we define decision variables $succ_j$ taking their values in $JOBS$, to indicate the successor of every job. The complete model formulation is the following,

$$\begin{aligned} & \text{minimize } \sum_{j \in JOBS} DUR_j + cleantime \\ & \forall j \in JOBS : succ_j \in JOBS \setminus \{j\} \\ & cleantime = cycle((succ_j)_{j \in JOBS}, (CLEAN_{ij})_{i,j \in JOBS}) \end{aligned}$$

where 'cycle' stands for the relation 'sequence into a single cycle without subcycles or repetitions'. The variable *cleantime* equals the total duration of the cycle.

3.10.2 Implementation

The Mosel model using the `cycle` constraint looks as follows.

```
model "B-5 Paint production (CP)"
uses "kalis"

setparam("DEFAULT_LB", 0)

declarations
  NJ = 5                                ! Number of paint batches (=jobs)
  JOBS=0..NJ-1

  DUR: array(JOBS) of integer           ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer    ! Cleaning times between jobs
  CB: array(JOBS) of integer            ! Cleaning times after a batch

  succ: array(JOBS) of cvar             ! Successor of a batch
  cleanTime,cycleTime: cvar             ! Durations of cleaning / complete cycle
end-declarations

initializations from 'Data/b5paint.dat'
  DUR CLEAN
end-initializations

forall(j in JOBS) do
  0 <= succ(j); succ(j) <= NJ-1; succ(j) <> j
end-do

! Assign values to 'succ' variables as to obtain a single cycle
! 'cleanTime' is the sum of the cleaning times
cycle(succ, cleanTime, CLEAN)

! Objective: minimize the duration of a production cycle
cycleTime = cleanTime +sum(j in JOBS) DUR(j)

! Solve the problem
if not cp_minimize(cycleTime) then
  writeln("Problem is infeasible")
  exit(1)
end-if
cp_show_stats
```

```

! Solution printing
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:\nBatch Duration Cleaning")
first:=1
repeat
  writeln("  ", first, sprintf(DUR(first),8),
          sprintf(CLEAN(first,getsol(succ(first))),9) )
  first:=getsol(succ(first))
until (first=1)

end-model

```

Notice that we have renumbered the tasks, starting the index range with 0, to conform with the input format expected by the `cycle` constraint.

3.10.3 Results

The optimal solution to this problem has a minimum cycle time of 243 minutes, resulting from 202 minutes of (incompressible) processing time and 41 minutes of cleaning.

The problem statistics produced by Xpress-Kalis for a model run reveal that the 'cycle' version of this model is the most efficient way of representing and solving the problem: it takes fewer nodes and a shorter execution time than the two versions of Section 3.8.

3.11 Generic binary constraints: Euler knight tour

Our task is to find a tour on a chessboard for a knight figure such that the knight moves through every cell exactly once and at the end of the tour returns to its starting point. The path of the knight must follow the standard chess rules: a knight moves either one cell vertically and two cells horizontally, or two cells in the vertical and one cell in the horizontal direction, as shown in the following graphic (Figure 3.3):

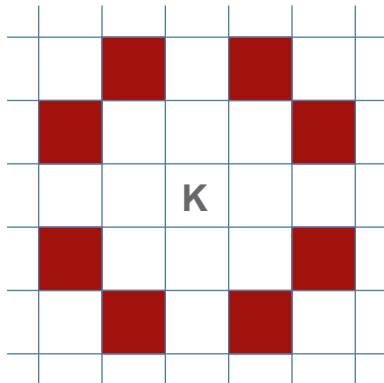


Figure 3.3: Permissible moves for a knight

3.11.1 Model formulation

To represent the chessboard we number the cells from 0 to $N-1$, where N is the number of cells of the board. The path of the knight is defined by N variables pos_i that indicate the i^{th} position of the knight on its tour.

The first variable is fixed to zero as the start of the tour. Another obvious constraint we need to state is that all variables pos_i take different values (every cell of the chessboard is visited exactly once):

$$all\text{-}different(pos_1, \dots, pos_N)$$

We are now left with the necessity to establish a constraint relation that checks whether consecutive positions define a valid knight move. To this aim we define a *new binary constraint* 'valid_knight_move' that checks whether a given pair of values defines a permissible move according to the chess rules for knight moves. Vertically and horizontally, the two values must be no more than two cells apart and the sum of the vertical and horizontal difference must be equal to three. The complete model then looks as follows.

$$\begin{aligned} &\forall i \in PATH = \{1, \dots, N\} : pos_i \in \{0, \dots, N-1\} \\ &pos_1 = 0 \\ &all_different(pos_1, \dots, pos_N) \\ &\forall i \in POS = \{1, \dots, N-1\} : valid_knight_move(pos_i, pos_{i+1}) \\ &valid_knight_move(pos_N, pos_1) \end{aligned}$$

3.11.2 Implementation

Testing whether moving from position a to position b is a valid move for a knight figure can be done with the following function *valid_knight_move* where 'div' means integer division without rest and 'mod' is the rest of the integer division:

```
function valid_knight_move(a,b)
  xa := a div E
  ya := a mod E
  xb := b div E
  yb := b mod E
  delta_x := |xa - xb|
  delta_y := |ya - yb|
  return ((delta_x ≤ 2) and (delta_y ≤ 2) and (delta_x + delta_y = 3))
end-function
```

The following Mosel model defines the user constraint function *valid_knight_move* as the implementation of the new binary constraints on pairs of $move_p$ variables (the constraints are established with *generic_binary_constraint*).

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

parameters
  S = 8                                ! Number of rows/columns
  NBSOL = 1                            ! Number of solutions sought
end-parameters

forward procedure print_solution(sol: integer)

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

declarations
  PATH = 1..N                          ! Cells on the chessboard
  pos: array(PATH) of cpvar             ! Cell at position p in the tour
end-declarations

! Fix the start position
pos(1) = 0

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

! The path of the knight 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
while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
end-do

! **** Test whether the move from position 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

!*****
! Solution printing
procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  forall(i in PATH)
    write(getval(pos(i)), if(i mod 10 = 0, "\n ", ""), " -> ")
  writeln("0")
end-procedure

end-model

```

The branching scheme used in this model is the `probe_assign_var` heuristic, in combination with the variable selection `KALIS_SMALLEST_MIN` (choose variable with smallest lower bound) and the value selection criterion `KALIS_MAX_TO_MIN` (from largest to smallest domain value). Another search strategy that was found to work well (though slower than the strategy in the code listing) is

```
cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, pos))
```

Our model defines two parameters. It is thus possible to change either the size of the chess-board (`S`) or the number of solutions sought (`NBSOL`) when executing the model without having to modify the model source.

3.11.3 Results

The first solution printed out by our model is the following tour.

```

0 -> 17 -> 34 -> 51 -> 36 -> 30 -> 47 -> 62 -> 45 -> 39
-> 54 -> 60 -> 43 -> 33 -> 48 -> 58 -> 52 -> 35 -> 41 -> 56
-> 50 -> 44 -> 38 -> 55 -> 61 -> 46 -> 63 -> 53 -> 59 -> 49
-> 32 -> 42 -> 57 -> 40 -> 25 -> 8 -> 2 -> 19 -> 4 -> 14
-> 31 -> 37 -> 22 -> 7 -> 13 -> 28 -> 18 -> 24 -> 9 -> 3
-> 20 -> 26 -> 16 -> 1 -> 11 -> 5 -> 15 -> 21 -> 6 -> 23
-> 29 -> 12 -> 27 -> 10 -> 0

```

3.11.4 Alternative implementation

Whereas the aim of the model above is to give an example of the definition of user constraints, it is possible to implement this problem in a more efficient way using the model developed for the cyclic scheduling problem in Section 3.10. The set of successors (domains of variables $succ_p$)

can be calculated using the same algorithm that we have used above in the implementation of the user-defined binary constraints.

Without repeating the complete model definition at this place, we simply display the model formulation, including the calculation of the sets of possible successors (procedure `calculate_successors`, and the modified procedure `print_sol` for solution printout. We use a simpler version of the 'cycle' constraints than the one we have seen in Section 3.10, its only argument is the set of successor variables—there are no weights to the arcs in this problem.

```

model "Euler Knight Moves"
  uses "kalis"

  parameters
    S = 8                                ! Number of rows/columns
    NBSOL = 1                            ! Number of solutions sought
  end-parameters

  forward procedure calculate_successors(p: integer)
  forward procedure print_solution(sol: integer)

  N:= S * S                                ! Total number of cells
  setparam("DEFAULT_LB", 0)
  setparam("DEFAULT_UB", N)

  declarations
    PATH = 0..N-1                        ! Cells on the chessboard
    succ: array(PATH) of cpvar           ! Successor of cell p
  end-declarations

  ! Calculate set of possible successors
  forall(p in PATH) calculate_successors(p)

  ! Each cell is visited once, no subtours
  cycle(succ)

  ! Search for up to NBSOL solutions
  solct:= 0
  while (solct<NBSOL and cp_find_next_sol) do
    solct+=1
    cp_show_stats
    print_solution(solct)
  end-do

  ! **** Calculate possible successors ****
  procedure calculate_successors(p: integer)
    declarations
      SuccSet: set of integer            ! Set of successors
    end-declarations

    xp := p div S
    yp := p mod S

    forall(q in PATH) do
      xq := q div S
      yq := q mod S
      delta_x := abs(xp-xq)
      delta_y := abs(yp-yq)
      if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
        SuccSet +={q}
      end-if
    end-do

    setdomain(succ(p), SuccSet)
  end-procedure

  !*****
  ! **** Solution printing ****
  procedure print_solution(sol: integer)
    writeln("Solution ", sol, ":")
    thispos:=0
    nextpos:=getval(succ(0))

```

```

ct:=1
while (nextpos<>0) do
  write(thispos, if(ct mod 10 = 0, "\n ", ""), " -> ")
  val:=getval(succ(thispos))
  thispos:=nextpos
  nextpos:=getval(succ(thispos))
  ct+=1
end-do
writeln("0")
end-procedure

end-model

```

The calculation of the domains for the $succ_p$ variables reduces these to 2-8 elements (as compared to the $N = 64$ values for every pos_p variables), which clearly reduces the search space for this problem.

This second model finds the first solution to the problem after 131 nodes taking just a fraction of a second to execute on a standard PC whereas the first model requires several thousand nodes and considerably longer running times. It is possible to reduce the number of branch-and-bound nodes even further by using yet another version of the 'cycle' constraint that works with successor and predecessor variables. This version of 'cycle' performs a larger amount of propagation, at the expense of (slightly) slower execution times for our problem. The procedure `calculate_successors` now sets the domain of $pred_p$ to the same values as $succ_p$ for all cells p .

```

declarations
  PATH = 0..N-1                                ! Cells on the chessboard
  succ: array(PATH) of cpvar                    ! Successor of cell p
  pred: array(PATH) of cpvar                    ! Predecessor of cell p
end-declarations

! Calculate sets of possible successors and predecessors
forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
cycle(succ, pred)

```

Figure 3.4 shows the graphical display of a knight's tour created with the 'user graph' functionality of IVE.

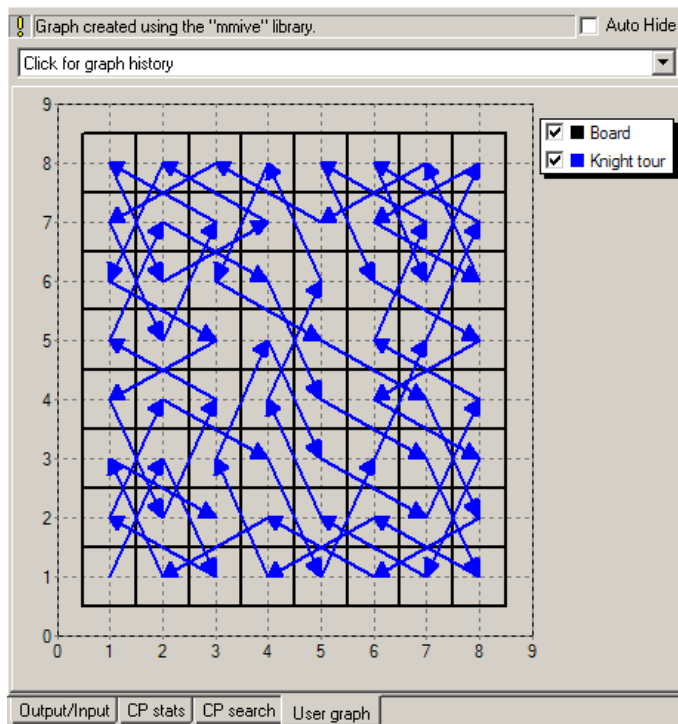


Figure 3.4: Knight's tour solution graph in IVE

3.11.5 Alternative implementation 2

Yet more efficient (a similar number of nodes with significantly shorter running times) is the following model version using the 'cycle' constraint described in Section 3.10. Since travel times from one position to the next are irrelevant in this model we simply fix all coefficients for the 'cycle' constraint to a constant and constrain the cycle length to the corresponding value.

All else, including the calculation of the sets of possible successors (procedure `calculate_successors`, and the procedure `print_sol` for solution printout is copied unchanged from the previous model.

```

model "Euler Knight Moves"
uses "kalis"

parameters
  S = 8                                ! Number of rows/columns
  NBSOL = 1                            ! Number of solutions sought
end-parameters

forward procedure calculate_successors(p: integer)
forward procedure print_solution(sol: integer)

N:= S * S                                ! Total number of cells
setparam("DEFAULT_LB", 0)
setparam("DEFAULT_UB", N)

declarations
  PATH = 0..N-1                        ! Cells on the chessboard
  succ: array(PATH) of cvar            ! Successor of cell p
end-declarations

! Calculate set of possible successors
forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
cycle(succ)

! Search for up to NBSOL solutions

```

```

solct:= 0
while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
end-do

! **** Calculate possible successors ****
procedure calculate_successors(p: integer)
  declarations
    SuccSet: set of integer          ! Set of successors
  end-declarations

  xp := p div S
  yp := p mod S

  forall(q in PATH) do
    xq := q div S
    yq := q mod S
    delta_x := abs(xp-xq)
    delta_y := abs(yp-yq)
    if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
      SuccSet +={q}
    end-if
  end-do

  setdomain(succ(p), SuccSet)
end-procedure

!*****
! **** Solution printing ****
procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  thispos:=0
  nextpos:=getval(succ(0))
  ct:=1
  while (nextpos<>0) do
    write(thispos, if(ct mod 10 = 0, "\n ", ""), " -> ")
    val:=getval(succ(thispos))
    thispos:=nextpos
    nextpos:=getval(succ(thispos))
    ct+=1
  end-do
  writeln("0")
end-procedure

end-model

```

Chapter 4

Enumeration

This chapter gives an overview on different issues related to the definition of search strategies with Xpress-Kalis, namely

- predefined search strategies,
- means of interrupting and restarting the enumeration,
- search callbacks, and
- the definition of user search strategies.

4.1 Predefined search strategies

With Xpress-Kalis a branching strategy is composed of three components:

- The *branching scheme* determines the shape of the search tree, this includes exhaustive schemes like `assign_var` and `split_domain` (enumeration of decision variables), `settle_disjunction` (enumeration over constraints), and `task_serialize` (enumeration of tasks in scheduling problems), or potentially incomplete searches with `probe_assign_var` or `probe_settle_disjunction`.
- The *variable selection* strategy determines the choice of the branching variables. Predefined selection criteria include

<code>KALIS_INPUT_ORDER</code>	variables in the given order,
<code>KALIS_LARGEST_MAX</code>	variable with largest upper bound,
<code>KALIS_LARGEST_MIN</code>	variable with largest lower bound,
<code>KALIS_MAX_DEGREE</code>	variable occurring in the largest number of constraints,
<code>KALIS_MAXREGRET_LB</code>	variable with largest difference between its lower bound and second-smallest domain value,
<code>KALIS_MAXREGRET_UB</code>	variable with largest difference between its upper bound and second-largest domain value,
<code>KALIS_RANDOM_VARIABLE</code>	random variable choice,
<code>KALIS_SDOMDEG_RATIO</code>	variable with smallest ratio domain size / degree,
<code>KALIS_SMALLEST_DOMAIN</code>	variable with smallest number of domain values/smallest domain interval,
<code>KALIS_SMALLEST_MAX</code>	variable with smallest upper bound,
<code>KALIS_SMALLEST_MIN</code>	variable with smallest lower bound).
<code>KALIS_WIDEST_DOMAIN</code>	variable with largest number of domain values/largest domain interval,

The user may also define his own variable selection strategy (see Section 4.4.3 below).

- The *value selection* strategy determines the choice of the branching value once the variable to be branched on is known. The following predefined selection criteria are available.

KALIS_MAX_TO_MIN	enumeration of values in decreasing order,
KALIS_MIDDLE_VALUE	enumerate first values in the middle of the variable's domain,
KALIS_MIN_TO_MAX	enumeration of values in increasing order,
KALIS_NEAREST_VALUE	choose the value closest to a target value previously specified with <code>settarget</code> ,
KALIS_RANDOM_VALUE	choose a random value out of the variable's domain.

The predefined criteria can be replaced by the user's own value selection strategy (see Section 4.4.3 below).

Depending on the choice of the branching scheme, it may be possible to specify additional parameters to configure the enumeration. The reader is referred to the Xpress-Kalis reference manual for further detail. In the case of constraint branching, there is quite obviously no variable or branching value selection. The special case of *task-based branching strategies* (branching scheme `task_serialize`) is discussed in Section 5.7. Enumeration of *continuous variables* (type `cpfloatvar`) always uses the branching scheme `split_domain` with the `KALIS_SMALLEST_DOMAIN` or `KALIS_WIDEST_DOMAIN` variable selection (the former is used by the default strategy, the latter often works better in purely continuous problems) and only a subset of the value selection strategies listed above.

Branching strategies may be defined for certain specified variables (or, where applicable, constraints or tasks), or, if the corresponding argument of the branching scheme function is left out, are applied to all decision variables in a model (see, for instance, model `b4seq_ka.mos` in Section 3.5).

If the user's model does not specify any branching strategy, then Xpress-Kalis will apply default strategies to enumerate all variables in the model. Even if one does not wish to change the default enumeration (for discrete variables: 'smallest domain first' and 'smallest value first'), it may still be desirable to define a branching strategy to fix a certain order for the enumeration of different groups of decision variables. The previous chapter contains several examples of this: in the model `a4sugar_ka.mos` (Section 3.6) we define an enumeration over some of the variables of a model, giving them thus preference over the remainder. In model `j5tax_ka.mos` (Section 3.9) we have seen an example of a search strategy composed of different enumerations for groups of decision variables.

If a model contains both, discrete and continuous variables (`cpvar` and `cpfloatvar`) the default strategies enumerate first the discrete and then the continuous variables.

4.2 Interrupting and restarting the search

When solving large applications it is often not possible to enumerate the complete search tree within a reasonable time span. Several *stopping criteria* are therefore available in Xpress-Kalis to interrupt the search. These are

MAX_BACKTRACKS	maximum number of backtracks,
MAX_COMPUTATION_TIME	limit on total time spent in search,
MAX_DEPTH	maximum search tree depth,
MAX_NODES	maximum number of nodes to explore,

MAX_SOLUTIONS	maximum number of solutions,
OPT_ABS_TOLERANCE	absolute difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization),
OPT_REL_TOLERANCE	relative difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization).

These parameters are accessed with the Mosel functions `setparam` and `getparam` (see, for example, the output of problem statistics in model `sudoku_ka.mos` in Section 3.3, and the search time limit set in the model `freqasgn.mos` of Section 3.4).

In optimization problems, after a solution has been found the search continues from this point unless the setting of parameter `OPTIMIZE_WITH_RESTART` is changed. The same is true if the search is interrupted by means of one of the above-named criteria and then continued, for instance with a different search strategy. To *restart the search* from the root node the procedure `cp_reset_search` needs to be called (as an example, see model `freqasgn.mos` in Section 3.4)

4.3 Callbacks

During the search the user's model may interact with the solver at certain predefined points by means of *callback functions*. This functionality is particularly useful to retrieve solution information for intermediate solutions during an optimization run as shown in the model `freqasgn.mos` (Section 3.4). Other than this *solution callback*, the user may set functions that will be called at every branch or at every node (see the Xpress-Kalis reference manual for further detail).

4.4 User-defined enumeration strategies

The following problem description is taken from Section 14.1 of '[Applications of optimization with Xpress-MP](#)'.

An operator needs to be assigned to each of the six machines in a workshop. Six workers have been pre-selected. Everyone has undergone a test of her productivity on every machine. Table 4.1 lists the productivities in pieces per hour. The machines run in parallel, that is, the total productivity of the workshop is the sum of the productivities of the people assigned to the machines.

Table 4.1: Productivity in pieces per hour

Workers	Machines					
	1	2	3	4	5	6
1	13	24	31	19	40	29
2	18	25	30	15	43	22
3	20	20	27	25	34	33
4	23	26	28	18	37	30
5	28	33	34	17	38	20
6	19	36	25	27	45	24

The objective is to determine an assignment of workers to machines that maximizes the total productivity. We may start by calculating a (non-optimal) heuristic solution using the following fairly natural method: choose the assignment $p \rightarrow m$ with the highest productivity, cross out

the line p and the column m (since the person has been placed and the machine has an operator), and restart this process until we have assigned all persons—the resulting assignment is highlighted in bold print in the data table. However, our aim really is to solve this problem to optimality for the case of parallel machines and also for machines working in series.

4.4.1 Model formulation

This problem type is well known under the name of the *assignment problem*. Let $PERS$ be the set of workers, $MACH$ the set of machines (both of the same size N), and $OUTP_{pm}$ the productivity of worker p on machine m . We define N integer variables $assign_p$ taking values in the set of machines, where $assign_p$ denotes the number of the machine to which the person p is assigned. The fact that a person p is assigned to a single machine m and a machine m is operated by a single person p is then expressed by the constraint that all variables $assign_p$ take different values.

$$\forall p \in PERS : assign_p \in MACH$$

$$all\text{-}different \left(\bigcup_{m \in MACH} assign_p \right)$$

Furthermore, let $output_p$ denote the output produced by person p . The values of these variables are obtained as discrete functions in the $assign_p$ variables:

$$\forall p \in PERS : output_p = OUTP_{p,assign_p}$$

4.4.1.1 Parallel machine assignment

The objective function to be maximized sums the $output_p$ variables.

$$\text{maximize } \sum_{p \in PERS} output_p$$

Certain assignments may be infeasible. In such a case, the value of the corresponding machine needs to be removed from the domain of the variable $assign_p$.

4.4.1.2 Machines working in series

If the machines work in series, the least productive worker on the machine she has been assigned to determines the total productivity of the workshop. An assignment will still be described by N variables $assign_p$ and an all-different constraint on these variables. We also have the $output_p$ variables with the constraints linking them to the values of the $assign_p$ variables from the previous model. To this we add a variable $pmin$ for the minimum productivity. The objective is to maximize $pmin$. This type of optimization problem where one wants to maximize a minimum is called *maximin*, or *bottleneck*.

$$\text{maximize } pmin$$

$$pmin = \text{minimum}_{p \in PERS}(output_p)$$

4.4.2 Implementation

The following Mosel program first implements and solves the model for the case of parallel machines. Afterwards, we define the variable $pmin$ that is required for solving the problem for the case that the machines work in series.

```
model "I-1 Personnel assignment (CP)"
uses "kalis"
```

```

forward procedure parallel_heur
forward procedure print_sol(text1,text2:string, objval:integer)

declarations
  PERS = 1..6                      ! Personnel
  MACH = 1..6                      ! Machines
  OUTP: array(PERS,MACH) of integer ! Productivity
end-declarations

initializations from 'Data/ilassign.dat'
  OUTP
end-initializations

! **** Exact solution for parallel machines ****

declarations
  assign: array(PERS) of cpvar      ! Machine assigned to a person
  output: array(PERS) of cpvar      ! Productivity of every person
  totalProd: cpvar                  ! Total productivity
  O: array(MACH) of integer         ! Auxiliary array for constraint def.
  Strategy: cpbranching             ! Branching strategy
end-declarations

forall(p in PERS) setdomain(assign(p), MACH)

! Calculate productivity per worker
forall(p in PERS) do
  forall(m in MACH) O(m) := OUTP(p,m)
  element(O, assign(p)) = output(p)
end-do

! Calculate total productivity
totalProd = sum(p in PERS) output(p)

! One person per machine
all_different(assign)

! Branching strategy
Strategy:= assign_var(KALIS_LARGEST_MAX, KALIS_MAX_TO_MIN, output)
cp_set_branching(Strategy)

! Solve the problem
if cp_maximize(totalProd) then
  print_sol("Exact solution (parallel assignment)", "Total", getsol(totalProd))
end-if

! **** Exact solution for serial machines ****

declarations
  pmin: cpvar                      ! Minimum productivity
end-declarations

! Calculate minimum productivity
pmin = minimum(output)

! Branching strategy
Strategy:= assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, output)
cp_set_branching(Strategy)

! Solve the problem
if cp_maximize(pmin) then
  print_sol("Exact solution (serial machines)", "Minimum", getsol(pmin))
end-if

```

When the solution to the parallel assignment problem is found, we print out the solution and re-start the search with a new branching strategy and a new objective function. Since the first search has finished completely (no interruption by a time limit, etc.) there is no need to reset the solver between the two runs.

The branching strategy chosen for the parallel assignment problem is inspired by the intuitive procedure described in the introduction to Section 4.4: instead of enumerating the possible assignments of workers to machines (= enumeration of the *assign_p* variables) we define an enu-

meration over the $output_p$ variables, choosing the variable with the largest remaining value (`KALIS_LARGEST_MAX`) and branch on its values in decreasing order (`KALIS_MAX_TO_MIN`). For the second problem we need to proceed differently: to avoid being left with a small productivity value for some worker p we pick first the $output_p$ variable with the smallest lower bound (`KALIS_SMALLEST_MIN`); again we enumerate the values starting with the largest one.

The following procedure `parallel_heur` may be added to the above program. It heuristically calculates a (non-optimal) solution to the parallel assignment problem using the intuitive procedure described in the introduction to Section 4.4.

```

procedure parallel_heur
  declarations
    ALLP, ALLM: set of integer      ! Copies of sets PERS and MACH
    HProd: integer                  ! Total productivity value
    pmax, omax, mmax: integer
  end-declarations

  ! Copy the sets of workers and machines
  forall(p in PERS) ALLP+={p}
  forall(m in MACH) ALLM+={m}

  ! Assign workers to machines as long as there are unassigned persons
  while (ALLP<>{}) do
    pmax:=0; mmax:=0; omax:=0

    ! Find the highest productivity among the remaining workers and machines
    forall(p in ALLP, m in ALLM)
      if OUP(p,m) > omax then
        omax:=OUP(p,m)
        pmax:=p; mmax:=m
      end-if

    assign(pmax) = mmax      ! Assign chosen machine to person pmax
    ALLP-={pmax}; ALLM-={mmax} ! Remove person and machine from sets
  end-do

  writeln("Heuristic solution (parallel assignment):")
  forall(p in PERS)
    writeln(" ",p, " operates machine ", getval(assign(p)),
           " (",getval(output(p)), " )")
  writeln(" Total productivity: ", getval(totalProd))
end-procedure

```

The model is completed with a procedure for printing out the solution in a properly formatted way.

```

procedure print_sol(text1,text2:string, objval:integer)
  writeln(text1,":")
  forall(p in PERS)
    writeln(" ",p, " operates machine ", getsol(assign(p)),
           " (",getsol(output(p)), " )")
  writeln(" ", text2, " productivity: ", objval)
end-procedure

end-model

```

4.4.3 User search

Instead of using predefined variable and value selection criteria as shown in all previous model implementations we may choose to define our own search heuristics. We now show how to implement the search strategies from the previous model ‘by hand.’ In our implementation we add each time a second criterion for breaking ties in cases where the main criterion applies to several variables at a time.

Variable selection: the variable selection function has a fixed format—it receives as its argument a list of variables of type `cpvarlist` and it returns an integer, the list index of the chosen branching variable. The list of variables passed into the function may contain variables that are already instantiated. As a first step in our implementation we, therefore, determine the

set `Vset` of yet uninstantiated variables—or to be more precise, the set of their indices in the list. The entries of the list of variables are accessed with the function `getvar`. Among the uninstantiated variables we calculate the set of variables `Iset` with the largest upper bound value (using function `getub`). Finally, among these variables, we choose the one with the smallest second-largest value (this corresponds to a *maximum regret* strategy). Predecessor (next-smallest) values in the domain of a decision variable are obtained with the function `getprev`. Inversely, we have function `getnext` for an enumeration of domain values in ascending order. The chosen index value is assigned to `returned` as the function's return value.

```
function varchoice(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
  end-declarations

  ! Set of uninstantiated variables
  forall(i in 1..getsize(Vars))
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if Vset={} then
    returned:= 0
  else
    ! Get the variable(s) with largest upper bound
    dmax:= max(i in Vset) getub(getvar(Vars,i))
    forall(i in Vset)
      if getub(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
    dmin:= dmax

    ! Choose variable with smallest next-best value among those indexed by 'Iset'
    forall(i in Iset) do
      prev:= getprev(getvar(Vars,i),dmax)
      if prev < dmin then
        returned:= i
        dmin:= prev
      end-if
    end-do
  end-if
end-function
```

The variable selection strategy `varchoicemin` for the second optimization run (serial machines) is implemented in a similar way. We first establish the set of variables with the smallest lower bound value (using `getlb`), `Iset`; among these we choose the variable with the smallest upper bound (`getub`).

```
function varchoicemin(Vars: cpvarlist): integer
  declarations
    Vset,Iset: set of integer
  end-declarations

  ! Set of uninstantiated variables
  forall(i in 1..getsize(Vars))
    if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if Vset={} then
    returned:= 0
  else
    ! Get the variable(s) with smallest lower bound
    dmin:= min(i in Vset) getlb(getvar(Vars,i))
    forall(i in Vset)
      if getlb(getvar(Vars,i)) = dmin then Iset+= {i}; end-if

    ! Choose variable with smallest upper bound among those indexed by 'Iset'
    dmax:= getparam("default_ub")
    forall(i in Iset)
      if getub(getvar(Vars,i)) < dmax then
        returned:= i
        dmax:= getub(getvar(Vars,i))
      end-if
    end-if
  end-if
end-function
```

Value selection: the value selection function receives as its argument the chosen branching variable and returns a branching value for this variable. The value selection criterion we have chosen (corresponding to `KALIS_MAX_TO_MIN`) is to enumerate all values for the branching variable, starting with the largest remaining one (that is, the variable's upper bound):

```
function valchoice(x: cpvar): integer
  returned:= getub(x)
end-function
```

Notice that with an `assign_var` or `assign_and_forbid` strategy, the user's value selection strategy should make sure to return a value that is currently in the branching variable's domain (a value chosen between the lower and upper bound is not guaranteed to lie in the domain) by using function `contains`.

Setting user search strategies: to indicate that we wish to use our own variable or value selection strategy we simply need to replace the predefined constants by the name of our Mosel functions:

```
Strategy:= assign_var("varchoice", "valchoice", output)
```

defines the strategy for the first optimization run and

```
Strategy:= assign_var("varchoicemin", "valchoice", output)
```

re-defines it for the serial machine case. Since our function `valchoice` does just the same as the `KALIS_MAX_TO_MIN` criterion, we could also combine it with our variable choice function:

```
Strategy:= assign_var("varchoicemin", KALIS_MAX_TO_MIN, output)
```

4.4.4 Results

The following table summarizes the results found with the different solution methods for the two problems of parallel and serial machines. There is a notable difference between the heuristic method and the exact solution to the problem with parallel machines.

Table 4.2: Optimal assignments for different model versions

	Alg.	Person						Productivity
		1	2	3	4	5	6	
Parallel Machines	Heur.	4 (19)	1 (18)	6 (33)	2 (26)	3 (34)	5 (45)	175
	Exact	3 (31)	5 (43)	4 (25)	6 (30)	1 (28)	2 (36)	193
Serial Machines	Exact	5 (40)	3 (30)	6 (33)	2 (26)	1 (28)	4 (27)	26

By adding output of solver statistics to our model (`cp_show_stats`) or consulting the *CP stats* display in IVE we find that our user search strategies result in the same search trees and program execution durations as with the predefined strategies for the parallel assignment and arrive at a slightly different solution for the serial case.

Chapter 5

Scheduling

This chapter shows how to

- define and setup the modeling objects `cptask` and `cpresource`,
- formulate and solve scheduling problems using these objects,
- access information from the modeling objects.

5.1 Tasks and resources

Scheduling and planning problems are concerned with determining a plan for the execution of a given set of tasks. The objective may be to generate a *feasible* schedule that satisfies the given constraints (such as sequence of tasks or limited resource availability) or to *optimize* a given criterion such as the makespan of the schedule.

Xpress-Kalis defines several aggregate modeling objects to simplify the formulation of standard scheduling problems: tasks (processing operations, activities) are represented by the type `cptask` and resources (machines, raw material etc.) by the type `cpresource`. When working with these scheduling objects it is often sufficient to state the objects and their properties, such as task duration or resource use; the necessary constraint relations are set up automatically by Xpress-Kalis (referred to as *implicit constraints*). In the following sections we show a number of examples using this mechanism:

- The simplest case of a scheduling problem involves only tasks and precedence constraints between tasks (project scheduling problem in Section 5.2).
- Tasks may be mutually exclusive, e.g. because they use the same unitary resource (disjunctive scheduling / sequencing problem in Section 5.3).
- Resources may be usable by several tasks at a time, up to a given capacity limit (cumulative resources, see Section 5.4).
- A different classification of resources is the distinction between renewable and non-renewable resources (see Section 5.5).
- Many extensions of the standard problems are possible, such as sequence-dependent setup time (see Section 5.6).

If the enumeration is started with the function `cp_schedule` the solver will employ specialized search strategies suitable for the corresponding (scheduling) problem type. It is possible to parameterize these strategies or to define user search strategies for scheduling objects (see Section 5.7). Alternatively, the standard optimization functions `cp_minimize` / `cp_maximize` may be used. In this case the enumeration does not exploit the structural information provided by the scheduling objects and works simply with decision variables.

The properties of scheduling objects (such as start time or duration of tasks) can be accessed and employed, for instance, in the definition of constraints, thus giving the user the possibility to extend the predefined standard problems with other types of constraints. For even greater flexibility Xpress-Kalis also enables the user to formulate his scheduling problems without the aggregate modeling objects, using dedicated global constraints on decision variables of type `cpvar`. Most examples in this chapter are therefore given with two different implementations, one using the scheduling objects and another without these objects.

5.2 Precedences

Probably the most basic type of a scheduling problem is to plan a set of tasks that are linked by precedence constraints.

The problem described in this section is taken from Section 7.1 ‘Construction of a stadium’ of the book ‘[Applications of optimization with Xpress-MP](#)’

A construction company has been awarded a contract to construct a stadium and wishes to complete it within the shortest possible time. Table 5.1 lists the major tasks and their durations in weeks. Some tasks can only start after the completion of certain other tasks, equally indicated in the table.

Table 5.1: Data for stadium construction

Task	Description	Duration	Predecessors
1	Installing the construction site	2	none
2	Terracing	16	1
3	Constructing the foundations	9	2
4	Access roads and other networks	8	2
5	Erecting the basement	10	3
6	Main floor	6	4,5
7	Dividing up the changing rooms	2	4
8	Electrifying the terraces	2	6
9	Constructing the roof	9	4,6
10	Lighting of the stadium	5	4
11	Installing the terraces	3	6
12	Sealing the roof	2	9
13	Finishing the changing rooms	1	7
14	Constructing the ticket office	7	2
15	Secondary access roads	4	4,14
16	Means of signalling	3	8,11,14
17	Lawn and sport accessories	9	12
18	Handing over the building	1	17

5.2.1 Model formulation

This problem is a classical project scheduling problem. We add a fictitious task with 0 duration that corresponds to the end of the project. We thus consider the set of tasks $TASKS = \{1, \dots, N\}$ where N is the fictitious end task.

Every construction task j ($j \in TASKS$) is represented by a task object $task_j$ with variable start time $task_j.start$ and a duration fixed to the given value DUR_j . The precedences between tasks are represented by a precedence graph with arcs (i, j) symbolizing that task i precedes task j .

The objective is to minimize the completion time of the project, that is the start time of the last, fictitious task N . We thus obtain the following model where an upper bound $HORIZON$ on the start times is given by the sum of all task durations:

tasks $task_j(j \in TASKS)$

$$\begin{aligned}
& \text{minimize } task_N.start \\
& \forall j \in TASKS : task_j.start \in \{0, \dots, HORIZON\} \\
& \forall j \in TASKS : task_j.duration = DUR_j \\
& \forall j \in TASKS : task_j.predecessors = \bigcup_{i \in TASKS, s.t. \exists ARC_{ij}} \{task_i\}
\end{aligned}$$

5.2.2 Implementation

The following model shows the implementation of this problem with Xpress-Kalis. Since there are no side-constraints, the earliest possible completion time of the schedule is the earliest start of the fictitious task N . To trigger the propagation of task-related constraints we call the function `cp_propagate`. At this point, constraining the start of the fictitious end task to its lower bound reduces all task start times to their feasible intervals through the effect of constraint propagation. The start times of tasks on the *critical path* are fixed to a single value. The subsequent call to minimization only serves for instantiating all variables with a single value so as to enable the graphical representation of the solution within IVE.

```

model "B-1 Stadium construction (CP)"
  uses "kalis"

  declarations
    N = 19                                ! Number of tasks in the project
                                          ! (last = fictitious end task)

    TASKS = 1..N
    ARC: array(range,range) of integer    ! Matrix of the adjacency graph
    DUR: array(TASKS) of integer           ! Duration of tasks
    HORIZON : integer                     ! Time horizon

    task: array(TASKS) of cptask           ! Tasks to be planned
    bestend: integer
  end-declarations

  initializations from 'Data/blstadium.dat'
    DUR ARC
  end-initializations

  HORIZON:= sum(j in TASKS) DUR(j)

! Setting up the tasks
forall(j in TASKS) do
  setdomain(getstart(task(j)), 0, HORIZON) ! Time horizon for start times
  set_task_attributes(task(j), DUR(j))     ! Duration
  setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
end-do                                     ! Precedences

if not cp_propagate then
  writeln("Problem is infeasible")
  exit(1)
end-if

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitious task N
bestend:= getlb(getstart(task(N)))
getstart(task(N)) <= bestend
writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
forall(j in TASKS) writeln(j, ": ", getstart(task(j)))

! Complete enumeration: schedule every task at the earliest possible date
result:= cp_minimize(getstart(task(N)))
forall(j in TASKS) writeln(j, ": ", getstart(task(j)))

end-model


```


Instead of indicating the predecessors of a task, we may just as well state the precedence constraints by indicating the sets of *successors* for every task:

```
setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
```

5.2.3 Results

The earliest completion time of the stadium construction is 64 weeks.

Users of IVE will notice that the execution of this model opens a special window, the *CP dashboard*, with a graphical display of the solution (see Figure 5.1). Hovering over the displayed chart will open pop-up boxes with detailed information about the tasks. Clicking on a task will display all task-related information in a separate window. It is possible to hide this window by clicking on the CP dashboard hide/unhide button .

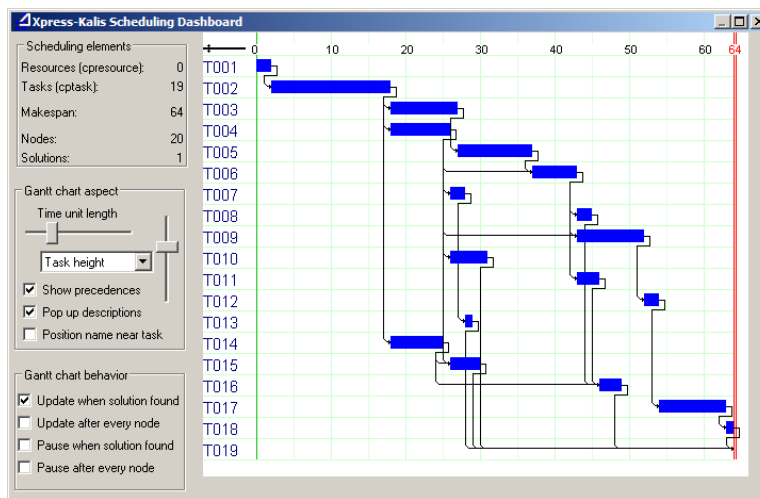


Figure 5.1: CP dashboard in IVE displaying the solution

5.2.4 Alternative formulation without scheduling objects

As for the previous formulation, we work with a set of tasks $TASKS = \{1, \dots, N\}$ where N is the fictitious end task. For every task j we introduce a decision variable $start_j$ to denote its start time. With DUR_j the duration of task j , the precedence relation 'task i precedes task j ' is stated by the constraint

$$start_i + DUR_i \leq start_j$$

We therefore obtain the following model for our project scheduling problem:

```
minimize start_N
forall j in TASKS : start_j in {0, ..., HORIZON}
forall i, j in TASKS, exists ARC_ij : start_i + DUR_i <= start_j
```

The corresponding Mosel model is printed in full below. Notice that we have used explicit posting of the precedence constraints—in the case of an infeasible data instance this may help tracing the cause of the infeasibility.

```
model "B-1 Stadium construction (CP)"
uses "kalis"

declarations
  N = 19                                     ! Number of tasks in the project
```

```

! (last = fictitious end task)
TASKS = 1..N
ARC: array(range,range) of integer ! Matrix of the adjacency graph
DUR: array(TASKS) of integer ! Duration of tasks
HORIZON : integer ! Time horizon

start: array(TASKS) of cpvar ! Start dates of tasks
bestend: integer
end-declarations

initializations from 'Data/blstadium.dat'
DUR ARC
end-initializations

HORIZON:= sum(j in TASKS) DUR(j)

forall(j in TASKS) do
    0 <= start(j); start(j) <= HORIZON
end-do

! Task i precedes task j
forall(i, j in TASKS | exists(ARC(i, j))) do
    Prec(i,j) := start(i) + DUR(i) <= start(j)
    if not cp_post(Prec(i,j)) then
        writeln("Posting precedence ", i, "-", j, " failed")
        exit(1)
    end-if
end-do

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitious task N
bestend:= getlb(start(N))
start(N) <= bestend
writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
forall(j in TASKS) writeln(j, ": ", start(j))

end-model

```

5.3 Disjunctive scheduling: unary resources

The problem of sequencing jobs on a single machine described in Section 3.5 may be represented as a disjunctive scheduling problem using the ‘task’ and ‘resource’ modeling objects.

The reader is reminded that the problem is to schedule the processing of a set of non-preemptive tasks (or jobs) on a single machine. For every task j its release date, duration, and due date are given. The problem is to be solved with three different objectives, minimizing the makespan, the average completion time, or the total tardiness.

5.3.1 Model formulation

The major part of the model formulation consists of the definition of the scheduling objects ‘tasks’ and ‘resources’.

Every job j ($j \in JOBS = \{1, \dots, NJ\}$) is represented by a task object $task_j$, with a start time $task_j.start$ in $\{REL_j, \dots, MAXTIME\}$ (where $MAXTIME$ is a sufficiently large value, such as the sum of all release dates and all durations, and REL_j the release date of job j) and the task duration $task_j.duration$ fixed to the given processing time DUR_j . All jobs use the same resource res of unitary capacity. This means that at most one job may be processed at any one time, we thus implicitly state the disjunctions between the jobs.

Another implicit constraint established by the task objects is the relation between the start, duration, and completion time of a job j .

$$\forall j \in JOBS : task_j.end = task_j.start + task_j.duration$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule) or, equivalently, to minimize the completion time *finish* of the last job. The complete model is then given by the following (where *MAXTIME* is a sufficiently large value, such as the sum of all release dates and all durations):

```

resource res
tasks taskj(j ∈ JOBS)
minimize finish
finish = maximumj ∈ JOBS(taskj. end)
res. capacity = 1
∀j ∈ JOBS : taskj. end ∈ {0, ..., MAXTIME}
∀j ∈ JOBS : taskj. start ∈ {RELj, ..., MAXTIME}
∀j ∈ JOBS : taskj. duration = DURj
∀j ∈ JOBS : taskj. requirementres = 1

```

Objective 2: The formulation of the second objective (minimizing the average processing time or, equivalently, minimizing the sum of the job completion times) remains unchanged from the first model—we introduce an additional variable *totComp* representing the sum of the completion times of all jobs.

```

minimize totComp
totComp = ∑j ∈ JOBS taskj. end

```

Objective 3: To formulate the objective of minimizing the total tardiness, we introduce new variables *late_j* to measure the amount of time that a job finishes after its due date. The value of these variables corresponds to the difference between the completion time of a job *j* and its due date *DUE_j*. If the job finishes before its due date, the value must be zero. The objective now is to minimize the sum of these tardiness variables:

```

minimize totLate
totLate = ∑j ∈ JOBS latej
∀j ∈ JOBS : latej ∈ {0, ..., MAXTIME}
∀j ∈ JOBS : latej ≥ taskj. end − DUEj

```

5.3.2 Implementation

The following implementation with Xpress-Kalis (file `b4seq3_ka.mos`) shows how to set up the necessary task and resource modeling objects. The resource capacity is set with procedure `set_resource_attributes` (the resource is of the type `KALIS_UNARY_RESOURCE` meaning that it processes at most one task at a time), for the tasks we use the procedure `set_task_attributes`. The latter exists in several overloaded versions for different combinations of arguments (task attributes)—the reader is referred to the Xpress-Kalis Reference Manual for further detail.

For the formulation of the `maximum` constraint we use an (auxiliary) list of variables: Xpress-Kalis does not allow the user to employ the access functions to modeling objects (`getstart`, `getduration`, etc.) in set expressions such as `union(j in JOBS) getend(task(j))`.

```

model "B-4 Sequencing (CP)"
uses "kalis"

forward procedure print_sol
forward procedure print_sol3

```

```

declarations
  NJ = 7                                ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer           ! Release dates of jobs
  DUR: array(JOBS) of integer           ! Durations of jobs
  DUE: array(JOBS) of integer           ! Due dates of jobs

  task: array(JOBS) of cptask           ! Tasks (jobs to be scheduled)
  res: cpresource                       ! Resource (machine)

  finish: cpvar                         ! Completion time of the entire schedule
end-declarations

initializations from 'Data/b4seq.dat'
  DUR REL DUE
end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations and disjunctions)
forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)

MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

forall(j in JOBS) do
  0 <= getstart(task(j)); getstart(task(j)) <= MAXTIME
  0 <= getend(task(j)); getend(task(j)) <= MAXTIME
end-do

! Start times
forall(j in JOBS) getstart(task(j)) >= REL(j)

!**** Objective function 1: minimize latest completion time ****
declarations
  L: cpvarlist
end-declarations

forall(j in JOBS) L += getend(task(j))
finish = maximum(L)

if cp_schedule(finish) >0 then
  print_sol
end-if

!**** Objective function 2: minimize average completion time ****
declarations
  totComp: cpvar
end-declarations

totComp = sum(j in JOBS) getend(task(j))

if cp_schedule(totComp) > 0 then
  print_sol
end-if

!**** Objective function 3: minimize total tardiness ****
declarations
  late: array(JOBS) of cpvar           ! Lateness of jobs
  totLate: cpvar
end-declarations

forall(j in JOBS) do
  0 <= late(j); late(j) <= MAXTIME
end-do

! Late jobs: completion time exceeds the due date
forall(j in JOBS) late(j) >= getend(task(j)) - DUE(j)

totLate = sum(j in JOBS) late(j)
if cp_schedule(totLate) > 0 then

```

```

        writeln("Tardiness: ", getsol(totLate))
        print_sol
        print_sol3
    end-if

!-----

! Solution printing
procedure print_sol
    writeln("Completion time: ", getsol(finish) ,
           " average: ", getsol(sum(j in JOBS) getend(task(j))))
    write("Rel\t")
    forall(j in JOBS) write(strfmt(REL(j),4))
    write("\nDur\t")
    forall(j in JOBS) write(strfmt(DUR(j),4))
    write("\nStart\t")
    forall(j in JOBS) write(strfmt(getsol(getstart(task(j))),4))
    write("\nEnd\t")
    forall(j in JOBS) write(strfmt(getsol(getend(task(j))),4))
    writeln
end-procedure

procedure print_sol3
    write("Due\t")
    forall(j in JOBS) write(strfmt(DUE(j),4))
    write("\nLate\t")
    forall(j in JOBS) write(strfmt(getsol(late(j)),4))
    writeln
end-procedure

end-model

```

5.3.3 Results

This model produces similar results as those reported for the model versions in Section 3.5. Figure 5.2 shows the Gantt chart display of the solution created by IVE. Above the Gantt chart we can see the resource usage display: the machine is used without interruption by the tasks, that is, even if we relaxed the constraints given by the release times and due dates it would not have been possible to generate a schedule terminating earlier.

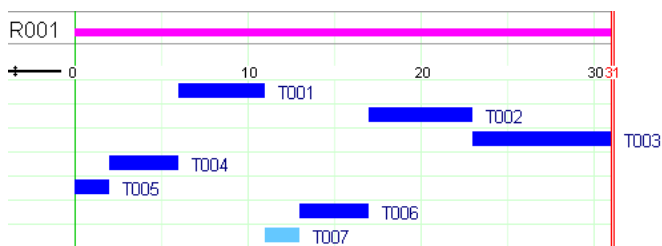


Figure 5.2: Solution display in IVE

5.4 Cumulative scheduling: discrete resources

The problem described in this section is taken from Section 9.4 ‘Backing up files’ of the book ‘Applications of optimization with Xpress-MP’

Our task is to save sixteen files of the following sizes: 46kb, 55kb, 62kb, 87kb, 108kb, 114kb, 137kb, 164kb, 253kb, 364kb, 372kb, 388kb, 406kb, 432kb, 461kb, and 851kb onto empty disks of 1.44Mb capacity. How should the files be distributed in order to minimize the number of floppy disks used?

5.4.1 Model formulation

This problem belongs to the class of *binpacking problems*. We show here how it may be formulated and solved as a cumulative scheduling problem, where the disks are the resource and the files the tasks to be scheduled.

The floppy disks may be represented as a single, discrete resource *disks*, where every time unit stands for one disk. The resource capacity corresponds to the disk capacity.

We represent every file f ($f \in FILES$) by a task object $file_f$, with a fixed duration of 1 unit and a resource requirement that corresponds to the given size $SIZE_f$ of the file. The 'start' field of the task then indicates the choice of the disk for saving this file.

The objective is to minimize the number of disks that are used, which corresponds to minimizing the largest value taken by the 'start' fields of the tasks (that is, the number of the disk used for saving a file). We thus have the following model.

```

resource disks
tasks  $file_f (f \in FILES)$ 
minimize diskuse
 $diskuse = \text{maximum}_{f \in FILES} (file_f.start)$ 
 $disks.capacity = CAP$ 
 $\forall f \in FILES : file_f.start \geq 1$ 
 $\forall f \in FILES : file_f.duration = 1$ 
 $\forall f \in FILES : file_f.requirement_{disks} = SIZE_f$ 

```

5.4.2 Implementation

The implementation with Xpress-Kalis is quite straightforward. We define a resource of the type `KALIS_DISCRETE_RESOURCE`, indicating its total capacity. The definition of the tasks is similar to what we have seen in the previous example.

```

model "D-4 Bin packing (CP)"
uses "kalis"

declarations
  ND: integer                ! Number of floppy disks
  FILES = 1..16              ! Set of files
  DISKS: range                ! Set of disks

  CAP: integer                ! Floppy disk size
  SIZE: array(FILES) of integer ! Size of files to be saved

  file: array(FILES) of cptask ! Tasks (= files to be saved)
  disks: cpresource            ! Resource representing disks
  L: cpvarlist
  diskuse: cpvar                ! Number of disks used
end-declarations

initializations from 'Data/d4backup.dat'
  CAP SIZE
end-initializations

! Provide a sufficiently large number of disks
ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
DISKS:= 1..ND

! Setting up the resource (capacity limit of disks)
set_resource_attributes(disks, KALIS_DISCRETE_RESOURCE, CAP)

! Setting up the tasks
forall(f in FILES) do
  setdomain(getstart(file(f)), DISKS)          ! Start time (= choice of disk)
  set_task_attributes(file(f), disks, SIZE(f)) ! Resource (disk space) req.
  set_task_attributes(file(f), 1)              ! Duration (= number of disks used)
end

```

```

end-do

! Limit the number of disks used
forall(f in FILES) L += getstart(file(f))
diskuse = maximum(L)

! Minimize the total number of disks used
if cp_schedule(diskuse) = 0 then
    writeln("Problem infeasible")
end-if

! Solution printing
writeln("Number of disks used: ", getsol(diskuse))
forall(d in 1..getsol(diskuse)) do
    write(d, ":")
    forall(f in FILES) write( if(getsol(getstart(file(f)))=d , " "+SIZE(f), ""))
    writeln(" space used: ",
            sum(f in FILES | getsol(getstart(file(f)))=d) SIZE(f))
end-do
cp_show_stats

end-model

```

5.4.3 Results

Running the model results in the solution shown in Table 5.2, that is, 3 disks are needed for backing up all the files.

Table 5.2: Distribution of files to disks

Disk	File sizes (in kb)	Used space (in Mb)
1	46 87 137 164 253 364 388	1.439
2	55 62 108 372 406 432	1.435
3	114 461 851	1.426

The visualization of the results in IVE is shown in Figure 5.3 with the resource usage profile at the top and the task Gantt chart in the lower half.

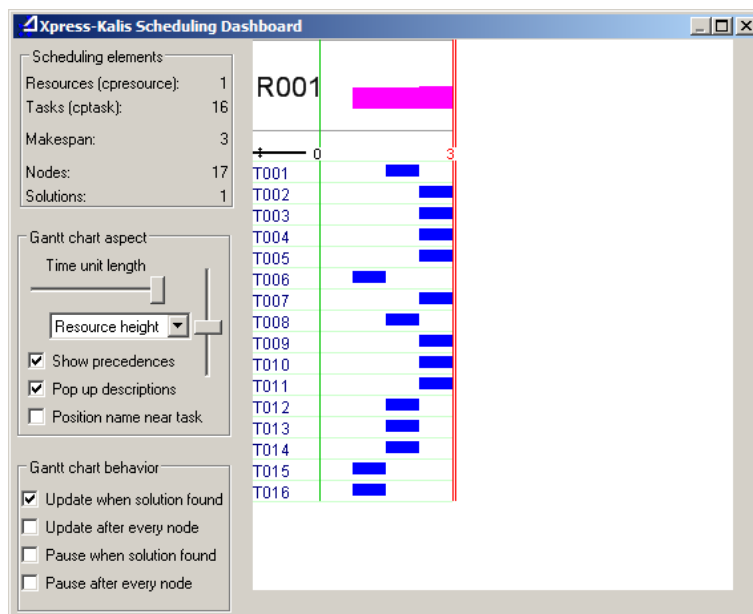


Figure 5.3: CP dashboard in IVE displaying the solution

5.4.4 Alternative formulation without scheduling objects

Instead of defining task and resource objects it is also possible to formulate this problem with a 'cumulative' constraint over standard finite domain variables that represent the different attributes of tasks without being grouped into a predefined object. A single 'cumulative' constraint expresses the problem of scheduling a set of tasks on one discrete resource by establishing the following relations between its arguments (five arrays of decision variables for the properties related to tasks—start, duration, end, resource use and size— all indexed by the same set R and a constant or time-indexed resource capacity):

$$\begin{aligned} \forall j \in R : start_j + duration_j &= end_j \\ \forall j \in R : use_j \cdot duration_j &= size_j \\ \forall t \in TIMES : \sum_{j \in R | t \in [UB(start_j) .. LB(end_j)]} use_j &\leq CAP_t \end{aligned}$$

where UB stands for 'upper bound' and LB for 'lower bound' of a decision variable.

Let $save_f$ denote the disk used for saving a file f and use_f the space used by the file ($f \in FILES$). As with scheduling objects, the 'start' property of a task corresponds to the disk chosen for saving the file, and the resource requirement of a task is the file size. Since we want to save every file onto a single disk, the 'duration' dur_f is fixed to 1. The remaining task properties 'end' and 'size' (e_f and s_f) that need to be provided in the formulation of 'cumulative' constraints are not really required for our problem; their values are determined by the other three properties.

5.4.5 Implementation

The following Mosel model implements the second model version using the `cumulative` constraint.

```
model "D-4 Bin packing (CP)"
uses "kalis", "mmsystem"

setparam("default_lb", 0)

declarations
  ND: integer                ! Number of floppy disks
  FILES = 1..16              ! Set of files
  DISKS: range               ! Set of disks

  CAP: integer               ! Floppy disk size
  SIZE: array(FILES) of integer ! Size of files to be saved

  save: array(FILES) of cpvar ! Disk a file is saved on
  use: array(FILES) of cpvar  ! Space used by file on disk
  dur,e,s: array(FILES) of cpvar ! Auxiliary arrays for 'cumulative'
  diskuse: cpvar              ! Number of disks used

  Strategy: array(FILES) of cpbranching ! Enumeration
  FSORT: array(FILES) of integer
end-declarations

initializations from 'Data/d4backup.dat'
  CAP SIZE
end-initializations

! Provide a sufficiently large number of disks
ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
DISKS:= 1..ND
finalize(DISKS)

! Limit the number of disks used
diskuse = maximum(save)

forall(f in FILES) do
  setdomain(save(f), DISKS) ! Every file onto a single disk
  use(f) = SIZE(f)
```



```

    dur(f) = 1
end-do

! Capacity limit of disks
cumulative(save, dur, e, use, s, CAP)

! Definition of search (place largest files first)
qsort(SYS_DOWN, SIZE, FSORT)      ! Sort files in decreasing order of size
forall(f in FILES)
    Strategy(f) := assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX, {save(FSORT(f))})
cp_set_branching(Strategy)

! Minimize the total number of disks used
if not cp_minimize(diskuse) then
    writeln("Problem infeasible")
end-if

! Solution printing
writeln("Number of disks used: ", getsol(diskuse))
forall(d in 1..getsol(diskuse)) do
    write(d, ":")
    forall(f in FILES) write( if(getsol(save(f))=d , " "+SIZE(f), ""))
writeln("    space used: ", sum(f in FILES | getsol(save(f))=d) SIZE(f))
end-do

end-model

```

The solution produced by the execution of this model has the same objective function value, but the distribution of the files to the disks is not exactly the same: this problem has several different optimal solutions, in particular those that may be obtained by interchanging the order numbers of the disks. To shorten the search in such a case it may be useful to add some *symmetry breaking* constraints that reduce the size of the search space by removing a part of the feasible solutions. In the present example we may, for instance, assign the biggest file to the first disk and the second largest to one of the first two disks, and so on, until we reach a lower bound on the number of disks required (a save lower bound estimate is given by rounding up to the next larger integer the sum of files sizes divided by the disk capacity).

5.5 Renewable and non-renewable resources

Besides the distinction ‘disjunctive–cumulative’ or ‘unary–discrete’ that we have encountered in the previous sections there are other ways of describing or classifying resources. Another important property is the concept of *renewable* versus *non-renewable* resources. The previous examples have shown instances of renewable resources (machine capacity, manpower etc.): the amount of resource used by a task is released at its completion and becomes available for other tasks. In the case of non-renewable resources (e.g. money, raw material, intermediate products), the tasks using the resource consume it, that is, the available quantity of the resource is diminished by the amount used up by processing a task.

Instead of using resources tasks may also *produce* certain quantities of resource. Again, we may have tasks that provide an amount of resource during their execution (renewable resources) or tasks that add the result of their production to a stock of resource (non-renewable resources).

Let us now see how to formulate the following problem: we wish to schedule five jobs P1 to P5 representing two stages of a production process. P1 and P2 produce an intermediate product that is needed by the jobs of the final stage (P3 to P5). For every job we are given its minimum and maximum duration, its cost or, for the jobs of the final stage, its profit contribution. There may be two cases, namely *model A*: the jobs of the first stage produce a given quantity of intermediate product (such as electricity, heat, steam) at every point of time during their execution, this intermediate product is consumed immediately by the jobs of the final stage. *Model B*: the intermediate product results as output from the jobs of the first stage and is required as input to start the jobs of the final stage. The intermediate product in model A is a renewable resource and in model B we have the case of a non-renewable resource.

5.5.1 Model formulation

Let $FIRST = \{P1, P2\}$ be the set of jobs in the first stage, $FINAL = \{P3, P4, P5\}$ the jobs of the second stage, and the set $JOBS$ the union of all jobs. For every job j we are given its minimum and maximum duration $MIND_j$ and $MAXD_j$ respectively. $RESAMT_j$ is the amount of resource needed as input or resulting as output from a job. Furthermore we have a cost $COST_j$ for jobs j of the first stage and a profit $PROFIT_j$ for jobs j of the final stage.

Model A (renewable resource)

The case of a renewable resource is formulated by the following model. Notice that the resource capacity is set to 0 indicating that the only available quantities of resource are those produced by the jobs of the first production stage.

```

resource res
tasks task_j(j ∈ JOBS)
maximize ∑_{j ∈ JOBS} (PROFIT_j - COST_j) × task_j.duration
res.capacity = 0
∀j ∈ JOBS : task_j.start, task_j.end ∈ {0, ..., HORIZON}
∀j ∈ JOBS : task_j.duration ∈ {MIND_j, ..., MAXD_j}
∀j ∈ FIRST : task_j.provision_res = RESAMT_j
∀j ∈ FINAL : task_j.requirement_res = RESAMT_j

```

Model B (non-renewable resource)

In analogy to the model A we formulate the second case as follows.

```

resource res
tasks task_j(j ∈ JOBS)
maximize ∑_{j ∈ JOBS} (PROFIT_j - COST_j) × task_j.duration
res.capacity = 0
∀j ∈ JOBS : task_j.start, task_j.end ∈ {0, ..., HORIZON}
∀j ∈ JOBS : task_j.duration ∈ {MIND_j, ..., MAXD_j}
∀j ∈ FIRST : task_j.production_res = RESAMT_j
∀j ∈ FINAL : task_j.consumption_res = RESAMT_j

```

However, this model does not entirely correspond to the problem description above since the production of the intermediate product occurs at the start of a task. To remedy this problem we may introduce an auxiliary task End_j for every job j in the first stage. The auxiliary job has duration 0, the same completion time as the original job and produces the intermediate product in the place of the original job.

```

∀j ∈ FIRST : task_End_j.end = task_j.end
∀j ∈ FIRST : task_End_j.duration = 0
∀j ∈ FIRST : task_j.production_res = 0
∀j ∈ FIRST : task_End_j.production_res = RESAMT_j

```

5.5.2 Implementation

The following Mosel model implements case A. We use the default scheduling solver (function

`cp_schedule`) indicating by the value `true` for the optional second argument that we wish to maximize the objective function.

```

model "Renewable resource"
uses "kalis", "mmsystem"

forward procedure solution_found

declarations
  FIRST = {'P1','P2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+FINAL

  MIND,MAXD: array(JOBS) of integer ! Limits on job durations
  RESAMT: array(JOBS) of integer ! Resource use/production
  HORIZON: integer ! Time horizon
  PROFIT: array(FINAL) of real ! Profit from production
  COST: array(JOBS) of real ! Cost of production
  CAP: integer ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask ! Task objects for jobs
  intermProd: cpresource ! Non-renewable resource (intermediate prod.)
end-declarations

initializations from 'Data/renewa.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
end-initializations

! Setting up resources
set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
setname(intermProd, "IntP")

! Setting up the tasks
forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
end-do

! Providing tasks
forall(j in FIRST) provides(task(j), RESAMT(j), intermProd)

! Requiring tasks
forall(j in FINAL) requires(task(j), RESAMT(j), intermProd)

! Objective function: total profit
totalProfit = sum(j in FINAL) PROFIT(j)*getduration(task(j)) -
              sum(j in JOBS) COST(j)*getduration(task(j))
cp_set_solution_callback("solution_found")
setparam("MAX_COMPUTATION_TIME", 30)

! Solve the problem
starttime:= gettime
if cp_schedule(totalProfit,true)=0 then
  exit(1)
end-if

! Solution printing
writeln("Total profit: ", getsol(totalProfit))
writeln("Job\tStart\tEnd\tDuration")
forall(j in JOBS)
  writeln(j, "\t ", getsol(getstart(task(j))), "\t ", getsol(getend(task(j))),
          "\t ", getsol(getduration(task(j))))

procedure solution_found
  writeln(gettime-starttime , " sec. Solution found with total profit = ",
          getsol(totalProfit))
  forall(j in JOBS)
    write(j, ": ", getsol(getstart(task(j))), "-", getsol(getend(task(j))),
          "(", getsol(getduration(task(j))), ")", " ")
  writeln
end-procedure

```

```
end-model
```

The model for case B adds the two auxiliary tasks (forming the set `ENDFIRST`) that mark the completion of the jobs in the first stage. The only other difference are the task properties `produces` and `consumes` that define the resource constraints. We only repeat the relevant part of the model:

```
declarations
  FIRST = {'P1','P2'}
  ENDFIRST = {'EndP1', 'EndP2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+ENDFIRST+FINAL

  MIND,MAXD: array(JOBS) of integer ! Limits on job durations
  RESAMT: array(JOBS) of integer ! Resource use/production
  HORIZON: integer ! Time horizon
  PROFIT: array(FINAL) of real ! Profit from production
  COST: array(JOBS) of real ! Cost of production
  CAP: integer ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask ! Task objects for jobs
  intermProd: cresource ! Non-renewable resource (intermediate prod.)
end-declarations

initializations from 'Data/renewb.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
end-initializations

! Setting up resources
set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
setname(intermProd, "IntP")

! Setting up the tasks
forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
end-do

! Production tasks
forall(j in ENDFIRST) produces(task(j), RESAMT(j), intermProd)
forall(j in FIRST) getend(task(j)) = getend(task("End"+j))

! Consumer tasks
forall(j in FINAL) consumes(task(j), RESAMT(j), intermProd)
```

5.5.3 Results

The behavior of the (default) search and the results of the two models are considerably different. The optimal solution with an objective of 344.9 for case B represented in Figure 5.5 is proven within a fraction of a second. Finding a good solution for case A takes several seconds on a standard PC; finding the optimal solution (see Figure 5.4) and proving its optimality requires several minutes of running time. The main reason for this poor behavior of the search is our choice of the objective function: the cost-based objective function does not propagate well and therefore does not help with pruning the search tree. A better choice for objective functions in scheduling problems generally are criteria involving the task decision variables (start, duration, or completion time, particularly the latter).

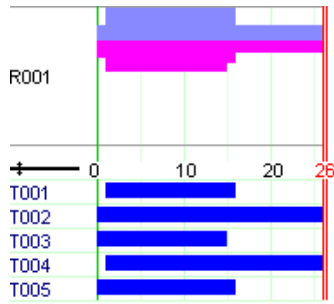


Figure 5.4: Solution for case A (resource provision and requirement)

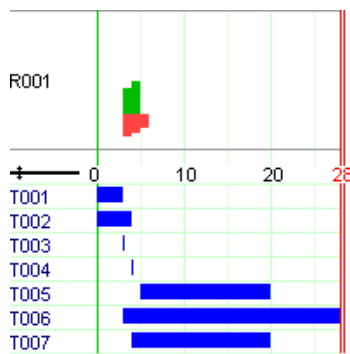


Figure 5.5: Solution for case B (resource production and consumption)

5.5.4 Alternative formulation without scheduling objects

Instead of defining task and resource objects we may equally formulate this problem with a ‘producer-consumer’ constraint over standard finite domain variables that represent the different attributes of tasks without being grouped into a predefined object. A single ‘producer-consumer’ constraint expresses the problem of scheduling a set of tasks producing or consuming a non-renewable resource by establishing the following relations between its arguments (seven arrays of decision variables for the properties related to tasks—start, duration, end, per unit and cumulated resource production, per unit and cumulated resource consumption— all indexed by the same set R):

$$\begin{aligned} \forall j \in R : start_j + duration_j &= end_j \\ \forall j \in R : produce_j \cdot duration_j &= psize_j \\ \forall j \in R : consume_j \cdot duration_j &= csize_j \\ \forall t \in TIMES : \sum_{j \in R | t \in [UB(start_j)..LB(end_j)]} (produce_j - consume_j) &\geq 0 \end{aligned}$$

where UB stands for ‘upper bound’ and LB for ‘lower bound’ of a decision variable.

5.5.5 Implementation

The following Mosel model implements the second model version using the `producer_consumer` constraint.

```
model "Non-renewable resource"
uses "kalis"

setparam("DEFAULT_LB", 0)
```

```

declarations
FIRST = {'P1','P2'}
ENDFIRST = {'EndP1','EndP2'}
FINAL = {'P3','P4','P5'}
JOBS = FIRST+ENDFIRST+FINAL
PCJOBS = ENDFIRST+FINAL

MIND,MAXD: array(JOBS) of integer ! Limits on job durations
RESAMT: array(JOBS) of integer ! Resource use/production
HORIZON: integer ! Time horizon
PROFIT: array(FINAL) of real ! Profit from production
COST: array(JOBS) of real ! Cost of production
CAP: integer ! Available resource quantity

totalProfit: cpfloatvar
fstart,fdur,fcomp: array(FIRST) of cpvar ! Start, duration & completion of jobs
start,dur,comp: array(PCJOBS) of cpvar ! Start, duration & completion of jobs
produce,consume: array(PCJOBS) of cpvar ! Production/consumption per time unit
psize,csize: array(PCJOBS) of cpvar ! Cumulated production/consumption
end-declarations

initializations from 'Data/renewb.dat'
[MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
end-initializations

! Setting up the tasks
forall(j in PCJOBS) do
    setname(start(j), j)
    setdomain(dur(j), MIND(j), MAXD(j))
    setdomain(comp(j), 0, HORIZON)
    start(j) + dur(j) = comp(j)
end-do
forall(j in FIRST) do
    setname(fstart(j), j)
    setdomain(fdur(j), MIND(j), MAXD(j))
    setdomain(fcomp(j), 0, HORIZON)
    fstart(j) + fdur(j) = fcomp(j)
end-do

! Production tasks
forall(j in ENDFIRST) do
    produce(j) = RESAMT(j)
    consume(j) = 0
end-do
forall(j in FIRST) fcomp(j) = comp("End"+j)

! Consumer tasks
forall(j in FINAL) do
    consume(j) = RESAMT(j)
    produce(j) = 0
end-do

! Resource constraint
producer_consumer(start, comp, dur, produce, psize, consume, csize)

! Objective function: total profit
totalProfit = sum(j in FINAL) PROFIT(j)*dur(j) -
               sum(j in FIRST) COST(j)*fdur(j)

if not cp_maximize(totalProfit) then
    exit(1)
end-if

writeln("Total profit: ", getsol(totalProfit))
writeln("Job\tStart\tEnd\tDuration")
forall(j in FIRST)
    writeln(j, "\t ", getsol(fstart(j)), "\t ", getsol(fcomp(j)),
            "\t ", getsol(fdur(j)))
forall(j in PCJOBS)
    writeln(j, "\t ", getsol(start(j)), "\t ", getsol(comp(j)),
            "\t ", getsol(dur(j)))

end-model

```

This model generates the same solution as the previous model version with a slightly longer running time (though still just a fraction of a second on a standard PC).

5.6 Extensions: setup times

Consider once more the problem of planning the production of paint batches introduced in Section 3.8. Between the processing of two batches the machine needs to be cleaned. The cleaning (or *setup*) times incurred are sequence-dependent and asymmetric. The objective is to determine a production cycle of the shortest length.

5.6.1 Model formulation

For every job j ($j \in JOBS = \{1, \dots, NJ\}$), represented by a task object $task_j$, we are given its processing duration DUR_j . We also have a matrix of cleaning times $CLEAN$ with entries $CLEAN_{jk}$ indicating the duration of the cleaning operation if task k succeeds task j . The machine processing the jobs is modeled as a resource res of unitary capacity, thus stating the disjunctions between the jobs.

With the objective to minimize the makespan (completion time of the last batch) we obtain the following model:

```
resource res
tasks task_j(j ∈ JOBS)
minimize finish
finish = maximum_{j ∈ JOBS}(task_j.end)
res.capacity = 1
∀j ∈ JOBS : task_j.duration = DUR_j
∀j ∈ JOBS : task_j.requirement_{res} = 1
∀j, k ∈ JOBS : setup(task_j, task_k) = CLEAN_{jk}
```

The tricky bit in the formulation of the original problem is that we wish to minimize the cycle time, that is, the completion of the last job plus the setup required between the last and the first jobs in the sequence. Since our task-based model does not contain any information about the sequence or rank of the jobs we introduce auxiliary variables *firstjob* and *lastjob* for the index values of the jobs in the first and last positions of the production cycle, and a variable *cleanlf* for the duration of the setup operation between the last and first tasks. The following constraints express the relations between these variables and the task objects:

```
firstjob, lastjob ∈ JOBS
firstjob ≠ lastjob
∀j ∈ JOBS : task_j.end = finish ⇔ lastjob = j
∀j ∈ JOBS : task_j.start = 1 ⇔ firstjob = j
cleanlf = CLEAN_{lastjob, firstjob}
```

Minimizing the cycle time then corresponds to minimizing the sum $finish + cleanlf$.

5.6.2 Implementation

The following Mosel model implements the task-based model formulated above. The setup times between tasks are set with the procedure `setsetuptimes` indicating the two task objects and the corresponding duration value.

```
model "B-5 Paint production (CP)"
uses "kalis"
```

```

declarations
  NJ = 5                                ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer           ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs

  task: array(JOBS) of cptask
  res: cpresource

  firstjob,lastjob,cleanlf,finish: cpvar
  L: cpvarlist
  cycleTime: cpvar                      ! Objective variable
  Strategy: array(range) of cpbranching
end-declarations

initializations from 'Data/b5paint.dat'
  DUR CLEAN
end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks
forall(j in JOBS) getstart(task(j)) >= 1                                ! Start times
forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)             ! Dur.s + disj.
forall(j,k in JOBS) setsetuptime(task(j), task(k), CLEAN(j,k), CLEAN(k,j))
                                                                    ! Cleaning times between batches

! Cleaning time at end of cycle (between last and first jobs)
setdomain(firstjob, JOBS); setdomain(lastjob, JOBS)
firstjob <> lastjob
forall(j in JOBS) equiv(getend(task(j))=getmakespan, lastjob=j)
forall(j in JOBS) equiv(getstart(task(j))=1, firstjob=j)
cleanlf = element(CLEAN, lastjob, firstjob)

forall(j in JOBS) L += getend(task(j))
finish = maximum(L)

! Objective: minimize the duration of a production cycle
cycleTime = finish - 1 + cleanlf

! Solve the problem
if cp_schedule(cycleTime) = 0 then
  writeln("Problem is infeasible")
  exit(1)
end-if
cp_show_stats

! Solution printing
declarations
  SUCC: array(JOBS) of integer
end-declarations

forall(j in JOBS)
  forall(k in JOBS)
    if getsol(getstart(task(k))) = getsol(getend(task(j)))+CLEAN(j,k) then
      SUCC(j):= k
      break
    end-if
  writeln("Minimum cycle time: ", getsol(cycleTime))
  writeln("Sequence of batches:\nBatch Start Duration Cleaning")
  forall(k in JOBS)
    writeln(" ", k, strfmt(getsol(getstart(task(k))),7), strfmt(DUR((k)),8),
      strfmt(if(SUCC(k)>0, CLEAN(k,SUCC(k)), getsol(cleanlf)),9))
  end-forall
end-model

```

5.6.3 Results

The results are similar to those reported in Section 3.8. It should be noted here that this model formulation is less efficient, in terms of search nodes and running times, than the previous

model versions, and in particular the ‘cycle’ constraint version presented in Section 3.10. However, the task-based formulation is more generic and easier to extend with additional features than the problem-specific formulations in the previous model versions.

The graphical representation with IVE looks as follows (Figure 5.6).

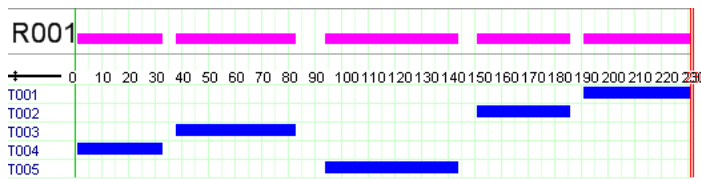


Figure 5.6: Solution graph in IVE

5.7 Enumeration

In the previous scheduling examples we have used the default enumeration for scheduling problems, invoked by the optimization function `cp_schedule`. The built-in search strategies used by the solver in this case are particularly suited if we wish to minimize the completion time of a schedule. With other objectives the built-in strategies may not be a good choice, especially if the model contains decision variables that are not part of scheduling objects (the built-in strategies always enumerate first the scheduling objects) or if we wish to *maximize* an objective. Xpress-Kalis makes it therefore possible to use the standard optimization functions `cp_minimize` and `cp_maximize` in models that contain scheduling objects, the default search strategies employed by these optimization functions being different from the scheduling-specific ones. In addition, the user may also define his own problem-specific enumeration as shown in the following examples.

User-defined enumeration strategies for scheduling problems may take two different forms: *variable-based* and *task-based*. The former case is the subject of Chapter 4, and we only give a small example here (Section 5.7.1). The latter will be explained with some more detail by the means of a job-shop scheduling example.

5.7.1 Variable-based enumeration

When studying the problem solving statistics for the bin packing problem in Section 5.4 we find that the enumeration using the default scheduling search strategies requires several hundred nodes to prove optimality. With a problem-specific search strategy we may be able to do better! Indeed, we shall see that a greedy-type strategy, assigning the largest files first, to the first disk with sufficient space is clearly more efficient.

5.7.1.1 Using `cp_minimize`

The only decisions to make in this problem are the assignments of files to disks, that is, choosing a value for the start time variables of the tasks. The following lines of Mosel code order the tasks in decreasing order of file sizes and define an enumeration strategy for their start times assigning each the smallest possible disk number first. Notice that the sorting subroutine `qsort` is defined by the module `mmsystem` that needs to be loaded with a `uses` statement at the beginning of the model.

```

declarations
  Strategy: array(range) of cpbranching
  FSORT: array(FILE) of integer
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)      ! Sort files in decreasing order of size
forall(f in FILE)

```

```

Strategy(f):=assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX,
                      {getstart(file(FSORT(f)))})
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if

```

The choice of the variable selection criterion (first argument of `assign_var`) is not really important here since every strategy $Strategy_f$ only applies to a single variable and hence no selection takes place. Equivalently, we may have written

```

declarations
  Strategy: cpbranching
  FSORT: array(FILE) of integer
  LS: cpvarlist
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)      ! Sort files in decreasing order of size
forall(f in FILES)
  LS+=getstart(file(FSORT(f)))
Strategy:=assign_var(KALIS_INPUT_ORDER, KALIS_MIN_TO_MAX, LS)
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if

```

With this search strategy the first solution found uses 3 disks, that is, we immediately find an optimal solution. The whole search terminates after 17 nodes and takes only a fraction of the time needed by the default scheduling or minimization strategies.

5.7.1.2 Using `cp_schedule`

The scheduling search consists of a pretreatment (*shaving*) phase and two main phases (`KALIS_INITIAL_SOLUTION` and `KALIS_OPTIMAL_SOLUTION`) for which the user may specify enumeration strategies by calling `cp_set_schedule_search` with the corresponding phase selection. The ‘initial solution’ phase aims at providing quickly a good solution whereas the ‘optimal solution’ phase proves optimality. Any search limits such as maximum number of nodes apply separately to each phase, an overall time limit (parameter `MAX_COMPUTATION_TIME`) only applies to the last phase.

The definition of variable-based branching schemes for the scheduling search is done in exactly the same way as what we have seen previously for standard search with `cp_minimize` or `cp_maximize`, replacing `cp_set_strategy` by `cp_set_schedule_strategy`:

```

cp_set_schedule_branching(KALIS_INITIAL_SOLUTION, Strategy)
if cp_schedule(diskuse)=0 then writeln("Problem infeasible"); end-if

```

With this search strategy, the optimal solution is found in the ‘initial solution’ phase after just 8 nodes and the enumeration stops there since the pretreatment phase has proven a lower bound of 3 which is just the value of the optimal solution.

NB: to obtain an output log from the different phases of the scheduling search set the control parameter `VERBOSE_LEVEL` to 2, that is, add the following line to your model before the start of the solution algorithm.

```

setparam("VERBOSE_LEVEL", 2)

```

5.7.2 Task-based enumeration

A task-based enumeration strategy consists in the definition of a selection strategy choosing the task to be enumerated, a value selection heuristic for the task durations, and a value selection heuristic for the task start times.

Consider the typical definition of a job-shop scheduling problem: we are given a set of jobs that each need to be processed in a fixed order by a given set of machines. A machine executes

one job at a time. The durations of the production tasks (= processing of a job on a machine) and the sequence of machines per job for a 6×6 instance are shown in Table 5.3. The objective is to minimize the makespan (latest completion time) of the schedule.

Table 5.3: 6×6 job-shop instance from [FM63]

Job	Machines						Durations					
1	3	1	2	4	6	5	1	3	6	7	3	6
2	2	3	5	6	1	4	8	5	10	10	10	4
3	3	4	6	1	2	5	5	4	8	9	1	7
4	2	1	3	4	5	6	5	5	5	3	8	9
5	3	2	5	6	1	4	9	3	5	4	3	1
6	2	4	6	1	5	3	3	3	9	10	4	1

5.7.2.1 Model formulation

Let $JOBS$ denote the set of jobs and $MACH$ ($MACH = \{1, \dots, NM\}$) the set of machines. Every job j is produced as a sequence of tasks $task_{jm}$ where $task_{jm}$ needs to be finished before $task_{j,m+1}$ can start. A task $task_{jm}$ is processed by the machine RES_{jm} and has a fixed duration DUR_{jm} .

The following model formulates the job-shop scheduling problem.

```

resources  $res_m(m \in MACH = \{1, \dots, NM\})$ 
tasks  $task_{jm}(j \in JOBS, m \in MACH)$ 
minimize  $finish$ 
 $finish = maximum_{j \in JOBS}(task_{j,NM}.end)$ 
 $\forall m \in MACH : res_m.capacity = 1$ 
 $\forall j \in JOBS, m \in MACH : task_{jm}.start, task_{jm}.end \in \{0, \dots, MAXTIME\}$ 
 $\forall j \in JOBS, m \in \{1, \dots, NM - 1\} : task_{jm}.successors = \{task_{j,m+1}\}$ 
 $\forall j \in JOBS, m \in MACH : task_{jm}.duration = DUR_{jm}$ 
 $\forall j \in JOBS, m \in MACH : task_{jm}.requirement_{RES_{jm}} = 1$ 

```

5.7.2.2 Implementation

The following Mosel model implements the job-shop scheduling problem and defines a task-based branching strategy for solving it. We select the task that has the smallest remaining domain for its start variable and enumerate the possible values for this variable starting with the smallest. A task-based branching strategy is defined in Xpress-Kalis with the function `task_serialize`, that takes as arguments the user task selection, value selection strategies for the duration and start variables, and the set of tasks it applies to. Such task-based branching strategies can be combined freely with any variable-based branching strategies.

```
- User branching strategy -
```

```
(c) 2008 Artelys S.A. and Fair Isaac Corporation
```

```
*****!)
```

```
model "Job shop (CP)"
```

```

uses "kalis", "mmsystem"

parameters

    DATAFILE = "jobshop.dat"

    NJ = 6                                ! Number of jobs
    NM = 6                                ! Number of resources
end-parameters

forward function select_task(tlist: cptasklist): integer

declarations

    JOBS = 1..NJ                          ! Set of jobs
    MACH = 1..NM                          ! Set of resources
    RES: array(JOBS,MACH) of integer      ! Resource use of tasks
    DUR: array(JOBS,MACH) of integer      ! Durations of tasks

    res: array(MACH) of cpresource        ! Resources
    task: array(JOBS,MACH) of cptask      ! Tasks
end-declarations

initializations from "Data/"+DATAFILE

    RES DUR
end-initializations

HORIZON:= sum(j in JOBS, m in MACH) DUR(j,m)
forall(j in JOBS) getend(task(j,NM)) <= HORIZON

! Setting up the resources (capacity 1)
forall(m in MACH)
    set_resource_attributes(res(m), KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations, resource used)
forall(j in JOBS, m in MACH)
    set_task_attributes(task(j,m), DUR(j,m), res(RES(j,m)))

! Precedence constraints between the tasks of every job
forall (j in JOBS, m in 1..NM-1)

```

```

! getstart(task(j,m)) + DUR(j,m) <= getstart(task(j,m+1))

setsuccessors(task(j,m), {task(j,m+1)})

! Branching strategy

Strategy:=task_serialize("select_task", KALIS_MIN_TO_MAX,
                          KALIS_MIN_TO_MAX,
                          union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
cp_set_branching(Strategy)

! Solve the problem

starttime:= gettime

if not cp_minimize(getmakespan) then
    writeln("Problem is infeasible")
    exit(1)
end-if

! Solution printing

cp_show_stats

write(gettime-starttime, "sec ")
writeln("Total completion time: ", getsol(getmakespan))

forall(j in JOBS) do
    write("Job ", strfmt(j,-2))

    forall(m in MACH | exists(task(j,m)))

        write(strfmt(RES(j,m),3), ":", strfmt(getsol(getstart(task(j,m))),3),
              "-", strfmt(getsol(getend(task(j,m))),2))

    writeln
end-do

!*****

! Task selection for branching

function select_task(tlist: cptasklist): integer

declarations
    Tset: set of integer
end-declarations

! Get the number of elements of "tlist"

```

```

listsize:= getsize(tlist)

! Set of uninstantiated tasks
forall(i in 1..listsize)

    if not is_fixed(getstart(gettask(tlist,i))) then

        Tset+= {i}

    end-if

returned:= 0

! Get a task with smallest start time domain
smin:= min(j in Tset) getsize(getstart(gettask(tlist,j)))

forall(j in Tset)

    if getsize(getstart(gettask(tlist,j))) = smin then

        returned:=j; break

    end-if

end-function

end-model

```

5.7.2.3 Results

An optimal solution to this problem has a makespan of 55. In comparison with the default *scheduling* strategy, our branching strategy reduces the number of nodes that are enumerated from over 300 to just 105 nodes with a comparable model execution time (however, for larger instances the default scheduling strategy is likely to outperform our branching strategy). The default *minimization* strategy does not find any solution for this problem within several minutes running time.

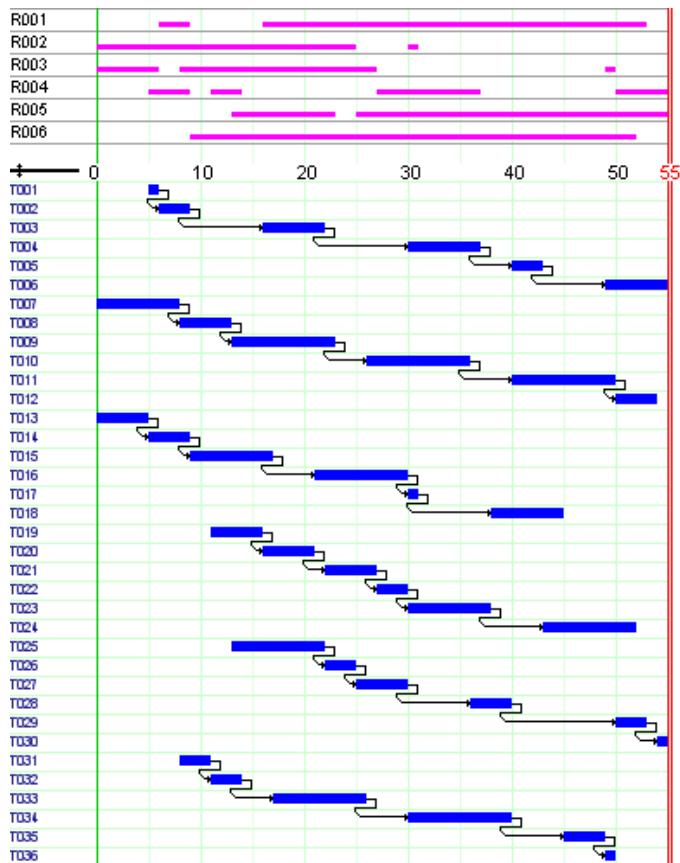


Figure 5.7: Solution graph in IVE

5.7.2.4 Alternative search strategies

Similarly to what we have seen above, we may define a user task selection strategy for the scheduling search. The only modifications required in our model are to replace `cp_set_` branching by `cp_set_schedule_strategy` and `cp_minimize` by `cp_schedule`. The definition of the user task choice function `select_task` remains unchanged.

```
! Branching strategy
Strategy:=task_serialize("select_task", KALIS_MIN_TO_MAX,
    KALIS_MIN_TO_MAX,
    union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
cp_set_schedule_strategy(KALIS_INITIAL_SOLUTION, Strategy)

! Solve the problem
if cp_schedule(getmakespan)=0 then
    writeln("Problem is infeasible")
    exit(1)
end-if
```

This strategy takes even fewer nodes for completing the enumeration than the standard search with user task selection.

Instead of the user-defined task selection function `select_task` it is equally possible to use one of the predefined task selection criteria:

`KALIS_SMALLEST_EST` / `KALIS_LARGEST_EST` choose task with the smallest/largest lower bound on its start time (*'earliest start time'*),

KALIS_SMALLEST_LST / KALIS_LARGEST_LST choose task with the smallest/largest upper bound on its start time (*'latest start time'*),

KALIS_SMALLEST_ECT / KALIS_LARGEST_ECT choose task with the smallest/largest lower bound on its completion time (*'earliest completion time'*),

KALIS_SMALLEST_LCT / KALIS_LARGEST_LCT choose task with the smallest/largest upper bound on its completion time (*'latest completion time'*).

For the present example, the best choice proves to be KALIS_SMALLEST_LCT (terminating the search after approximately 60 nodes with both `cp_schedule` and `cp_minimize`):

```
Strategy:=task_serialize(KALIS_SMALLEST_LCT, KALIS_MIN_TO_MAX,  
    KALIS_MIN_TO_MAX,  
    union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
```

5.7.3 Choice of the propagation algorithm

The performance of search algorithms for scheduling problems relies not alone on the definition of the enumeration strategy; the choice of the propagation algorithm for resource constraints may also have a noticeable impact. The propagation type is set by an optional last argument of the procedure `set_resources_attributes`, such as

```
set_resource_attributes(res, KALIS_DISCRETE_RESOURCE, CAP, ALG)
```

where `ALG` is the propagation algorithm choice.

For *unary resources* we have a choice between the algorithms `KALIS_TASK_INTERVALS` and `KALIS_DISJUNCTIONS`. The former achieves stronger pruning at the cost of a larger computational overhead making the choice of `KALIS_DISJUNCTIONS` more competitive for small to medium sized problems. Taking the example of the jobshop problem from the previous section, when using `KALIS_DISJUNCTIONS` the default scheduling search is about 4 times faster than with `KALIS_TASK_INTERVALS` although the number of nodes explored is slightly larger than with the latter.

The propagation algorithm options for *cumulative resources* are `KALIS_TASK_INTERVALS` and `KALIS_TIMETABLING`. The filtering algorithm `KALIS_TASK_INTERVALS` is stronger and relatively slow making it the preferred choice for hard, medium-sized problems whereas `KALIS_TIMETABLING` should be given preference for very large problems where the computational overhead of the former may be prohibitive. In terms of an example, the binpacking problem we have worked with in Sections 5.4 and 5.7.1 solves about three times faster with `KALIS_TASK_INTERVALS` than with `KALIS_TIMETABLING` (using the default scheduling search).

II. Xpress-Kalis extensions

Chapter 6

Implementing Kalis extensions

Xpress-Kalis makes accessible a subset of the functionality of the Artelys Kalis library within Mosel. However, sometimes industrial applications give rise to specific constraints that are not easily expressed with the constraint relations of Xpress-Kalis or for which more efficient propagation algorithms are known. In such a case it may be worthwhile to implement a new constraint relation for the specific problem at hand, using the functionality of *Xpress-Kalis extensions* (i.e., writing an add-on C++ library). Besides a more straightforward model formulation (and hence higher readability and easier maintenance of the model), implementing a specific constraint propagation algorithm might have a considerable impact on the solving times. The example of constraint definition given in this chapter being very simple, the reader is invited to take a look at the more realistic examples in the examples database on the [Xpress website](#) to confirm this statement.

The definition of user constraints often also calls for an enumeration tailored to the particular problem (obviously, the implementation of user branching strategies is not conditioned by the definition of new constraints). In the previous chapters we have seen examples of user branching strategies implemented directly in the Mosel model via the definition of variable and value selection subroutines. The same user strategies can be implemented in an Xpress-Kalis extension, the latter being preferable if the variable or value selection relies on (the repeated execution of) some computationally expensive algorithm. Later in this chapter we shall see an example of a user search strategy comparing the two implementation options. As a major difference to the implementation within the Mosel language the implementation by an Xpress-Kalis extension also makes it possible to define an entirely new *branching scheme*, potentially even involving other objects than the variables, (disjunctive) constraints, or tasks branched on in the existing schemes.

Note: this part is addressed at expert users, it deals with functionality on a lower level than the previous chapters. It requires more profound understanding of Constraint Programming techniques and of the functioning of a constraint solver, in particular the Artelys Kalis Library. Furthermore the reader is expected to have some experience with programming in C++.

6.1 Software architecture

The following graphic represents the standard distribution of Mosel + Xpress-Kalis so far. Xpress-Kalis was provided in the form of a Mosel module, *kalis*, that included the Artelys Kalis Library. The *kalis* module interacts with the Mosel Native Interface (Mosel NI) to give access to functionality of the Artelys Kalis library within the Mosel language.

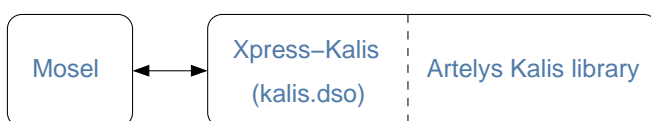


Figure 6.1: Standard distribution of Mosel + Xpress-Kalis

The software architecture for Xpress-Kalis extensions looks somewhat different. Most importantly, the Artelys Kalis Library is now provided separately making its functionality directly accessible to the extension libraries. From the (Mosel) user point of view nothing changes, the *kalis* module provides the Xpress-Kalis functionality in the Mosel language just in the same way as this has been the case with previous releases.

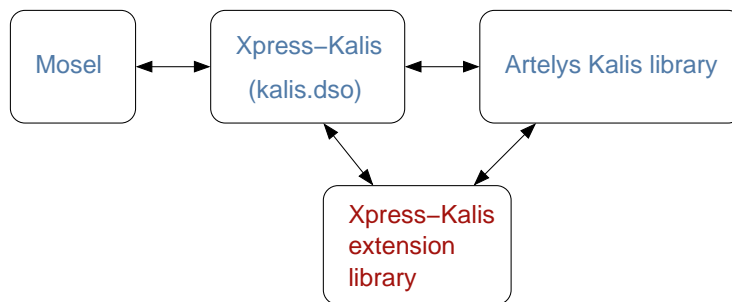


Figure 6.2: Xpress-Kalis extensions architecture

6.2 Required software and installation

Kalis extensions are implemented by writing a C++ program that (a) defines new classes derived from some classes of the Kalis Library and (b) follows the conventions of the Mosel Native Interface for the definition of subroutines. Besides the standard installation of Mosel with Xpress-Kalis you therefore need the following:

1. the Mosel Native Interface
2. the Xpress-Kalis extensions package
3. a C/C++ compiler

The *Mosel Native Interface (NI)* is included in the standard Mosel distribution and should be readily installed on your computer if you have followed the standard Xpress-MP installation procedure. The functions of the Mosel NI are documented in the NI Reference Manual, and the NI User Guide describes several examples of module definitions. Both documents are part of the Mosel documentation in your Xpress-MP installation and they are equally available for download from the [Xpress website](#).

The *Xpress-Kalis extensions package* contains the header files for the Xpress-Kalis module and the Artelys Kalis library that are necessary if you wish to compile extension libraries. It also includes a set of extension examples a selection of which are documented in the remainder of this chapter. **NB:** There is no need to install the extensions package for standard use of Xpress-Kalis or if you simply wish to use readily compiled extension libraries.

The *Artelys Kalis library* itself is provided with the standard Xpress-Kalis distribution. The installation procedure sets up all the required environment variables. For examples of working with this library, and in particular how to use the extension features of the Kalis library, the reader is referred to the extract of the Artelys Kalis User Guide provided with the extensions package.

Our examples use ANSI C/C++. The Artelys Kalis library requires under Windows the MSVC6.0 C++ compiler (or more recent versions) and under Linux gcc version 3.4.6.

6.3 Compilation

All user extensions need to include the header files of the three software components used by them: the Kalis library, the Mosel NI, and the extensions interface of Xpress-Kalis.

```
#include "Kalis.h"
#include "xprm_ni.h"
#include "XpressKalisExtensions.h"
```

Makefiles for the corresponding platform and operating system are provided with the examples of the Xpress-Kalis extensions package (see the examples subdirectory `kalis/Extensions` of your Xpress-MP installation). Once the extension library *myExtension.dll* (or *myExtension.so* for Linux/Unix) has been created it needs to be copied into the subdirectory `dso` of the Xpress-MP installation or any directory pointed at by the environment variable `MOSEL_DSO`. Xpress-Kalis will automatically load any extension libraries encountered in these locations.

Chapter 7

Writing a user constraint

The constraint relation 'DIFFEZ' described in this section has originally been developed for a personnel planning application in a disabled care center (see []).

Three working locations are to be staffed from a pool of five persons (A, B, C, D, E), each location taking 1 2 persons. A member of staff is either assigned to a working location or has a day off. There are also some specific constraints for certain persons:

1. B is accompanied/supervised by D.
2. C only works at location 1.
3. E can only work at locations 1 or 2 if A is assigned to the same post.
4. A cannot work at the same location as B or C.

We wish to generate the full list of all possible work plans.

7.1 Model formulation

The working location assigned to a person p from the set of staff $STAFF$ is represented by the decision variable $assign_p$. These decision variables take values in the set of working locations $POSTS = \{0, 1, \dots, NL\}$ where $1, \dots, NL$ are the actual working locations and 0 stands for 'day off'. Every location l has a given minimum and maximum staff level ($MINSTAFF_l$ and $MAXSTAFF_l$).

Relations (1) and (2) are expressed by the constraints

$$\begin{aligned} assign_B \geq 1 &\Rightarrow assign_B = assign_D \\ assign_C &\leq 1 \end{aligned}$$

and the constraint (3) can be formulated by the following two implications.

$$\begin{aligned} assign_E = 1 &\Rightarrow assign_A = 1 \\ assign_E = 2 &\Rightarrow assign_A = 2 \end{aligned}$$

The relation 'A not at the same location as B or C' requires some special care given that although they must not work together A might well have time off simultaneously with B or C.

$$\begin{aligned} assign_A &\neq assign_B \text{ or } (assign_A = 0 \text{ and } assign_B = 0) \\ assign_A &\neq assign_C \text{ or } (assign_A = 0 \text{ and } assign_C = 0) \end{aligned}$$

The staffing limits on the working locations:

$$\forall l \in POSTS : MINSTAFF_l \leq |assign_p = l|_{p \in STAFF} \leq MAXSTAFF_l$$

are best expressed by a 'distribute' (global cardinality) relation as described in Section 3.7 that establishes the lower and upper staffing limits for all working locations at once. We obtain the following model:

$$\begin{aligned} \forall p \in \text{STAFF} : \text{assign}_p &\in \text{POSTS} = \{0, 1, \dots, \text{NL}\} \\ \text{assign}_B \geq 1 &\Rightarrow \text{assign}_B = \text{assign}_D \\ \text{assign}_C &\leq 1 \\ \text{assign}_E = 1 &\Rightarrow \text{assign}_A = 1 \\ \text{assign}_E = 2 &\Rightarrow \text{assign}_A = 2 \\ \text{assign}_A \neq \text{assign}_B &\text{ or } (\text{assign}_A = 0 \text{ and } \text{assign}_B = 0) \\ \text{assign}_A \neq \text{assign}_C &\text{ or } (\text{assign}_A = 0 \text{ and } \text{assign}_C = 0) \\ \text{distribute}(\text{assign}_{\text{STAFF}}, \text{POSTS}, \text{MINSTAFF}_{\text{POSTS}}, \text{MAXSTAFF}_{\text{POSTS}}) \end{aligned}$$

7.2 Implementation: Mosel model

The following Mosel implementation of our model (file `diffez.mos`) indicates two different formulations for the relations (4) 'A not at the same location as B or C': a straightforward implementation as composite constraint relations using the logical 'and' and 'or' connectors or alternatively, by using a new constraint relation `diffez`. The implementation of this new constraint as a user extension to Xpress-Kalis is described in the following section.

```
model "Personnel planning using DIFFEZ"
uses "kalis"

declarations
  NL = 3
  POSTS = 0..NL ! 1-NL: working locations, 0: time off
  STAFF = {'A','B','C','D','E'} ! Personnel
  MINSTAFF,MAXSTAFF: array(POSTS) of integer
  assign: array(STAFF) of cpvar ! Working place assignments
end-declarations

MINSTAFF::(0..NL) [0,1,1,1]
MAXSTAFF::(0..NL) [2,2,2,2]

forall(p in STAFF) setdomain(assign(p), POSTS)

! B accompanied by D
implies(assign('B')>=1, assign('B') = assign('D'))

! C only at location 1 (or time off)
assign('C') <= 1

! E at locations 1 or 2 only if A also assigned to it
implies(assign('E')=1, assign('A')=1)
implies(assign('E')=2, assign('A')=2)

! A not with B or C
diffez(assign('A'),assign('B'))
diffez(assign('A'),assign('C'))

(! Equivalent constraint formulation:
  assign('A')<>assign('B') or (assign('A')=0 and assign('B')=0)
  assign('A')<>assign('C') or (assign('A')=0 and assign('C')=0)
!)

! Lower and upper staffing limits for all working locations (incl. time off)
distribute(assign, POSTS, MINSTAFF, MAXSTAFF)

ct:=0
write("      Person: ")
forall(p in STAFF) write(" ", p)
while(cp_find_next_sol) do
  ct+=1
  write("\nSolution ", strfmt(ct,2), ": ")
endwhile
```

```

        forall(p in STAFF) write(" ", assign(p).sol)
    end-do
    writeln
end-model

```

This model produces a list of 18 different work plans.

7.3 Implementation: user extension

A user extension for Xpress-Kalis is a (user-written) C++ library following certain conventions as explained subsequently. An extension library adds to the functionality of the Artelys Kalis solver by defining new (derived) classes for constraints or—as we shall see in the next section—branching schemes. The Mosel module *kalis* loads any extensions identified as such and makes their functionality visible within Mosel.

From a Mosel point of view, user extensions always come in the form of subroutines (entirely new ones or overloaded versions to existing subroutines). As a quick test whether or which extensions are present you may display the list of functions of the module *kalis* using Mosel's `exam` command, *i.e.*, from the command line

```
mosel -c "exam -s kalis"
```

or within IVE select the menu *Modules* \gg *List available modules* to open the module display window, there choose *kalis* and the *Functions & Procedures* tab. The resulting list of Xpress-Kalis subroutines includes all subroutines from extension libraries identified by the *kalis* module.

The implementation of an Xpress-Kalis extension library has four major components:

1. the declaration of the new subroutine(s) for the Mosel language,
2. the definition of the four Xpress-Kalis interface functions,
3. the implementation of the C function(s) associated with the entries in the list of Mosel subroutines,
4. the implementation of the C++ class(es) and any related methods as required by the extension mechanisms of the Artelys Kalis library.

For the implementation of the 'DIFFEZ' relation we have a single entry in the Mosel subroutine table (defining function `diffez`) and correspondingly, the definition of a single C function associated with this entry. On the Kalis side, there is the definition of a new class derived from the Kalis class `KUserConstraint` including a set of methods as required by the solver. We shall now take a closer look at each of these components.

7.3.1 List of Mosel subroutines

The list of Mosel subroutines declares the prototypes of the new subroutines for the Mosel language defined by the extension and establishes the correspondence with the C functions implementing the Mosel subroutines. The list has a standardized format defined by the Mosel Native Interface (NI); this format is documented in the 'Mosel NI Reference manual' and the 'Mosel NI User Guide' contains several examples of lists of subroutines.

For the 'DIFFEZ' constraint we have the following list of subroutines with a single entry:

```

static const int numNewFct = 1; // No. of new functions in this extension

static XPRMdsOfct extensions[numNewFct] =
{
    {
        "diffez", // name of the function within Mosel

```

```

10001,                // code (unique value within this extension)
XPRM_TYP_EXTN,        // return type (here: CP constraint, an "external type")
2,                    // number of arguments
"cpctr:|cpvar|cpvar|", // signature (external type names in arguments
                        // are delimited by "|")
dez_diffiezConstraint // function called for constraint creation
}
};

```

One entry of the list of subroutines consists of (in this order):

- The *name* of the subroutine that will be used within the Mosel language (**NB:** the subroutine name is case sensitive, if you wish to define, for instance, lower and upper case versions, you need to specify two separate entries in the list of subroutines).
- A *reference number*: an integer value that should be unique within the extension; Xpress-Kalis dynamically renumbers all subroutines from extensions each time the *kalis* module is loaded to prevent clashes among extensions or between an extension and the Xpress-Kalis subroutines.
- The *return type* (if the subroutine is a function): for Xpress-Kalis extensions this usually is an external type (code `XPRM_TYP_EXTN`).
- The *signature* of the Mosel subroutine: the return type (if the subroutine is a function), followed by a colon and the types of the arguments. The names of external types in the arguments must be surrounded by `"`, for the encoding of Mosel's own types and structured data (arrays, sets, lists) please refer to the Mosel NI Reference manual.
- The *C function* implementing the Mosel subroutine (see Section 7.3.3 below).

Within IVE you may use the *Module wizzard* to generate a skeleton template for the list of subroutines and the required implementation functions in the Mosel NI format. Attention: the template file created by the wizzard will contain a module initialization function (and some associated data structures), this function is not required in the present case and must be deleted.

7.3.2 Xpress-Kalis interface functions

Every extension library must define the four predefined Xpress-Kalis interface functions. These functions are called by Xpress-Kalis to exchange information with the extension library and most importantly, they are used to identify a library found on the *DSO search path* (the current directory, subdirectory `dso` of the Xpress-MP installation, and any locations defined by the environment variable `MOSEL_DSO`) as an Xpress-Kalis extension.

```

#ifdef _WIN32
#define EXTDLLIMPORTEXPOT __declspec(dllexport)
#else
#define EXTDLLIMPORTEXPOT
#endif

// Return the list of extensions in this module
extern "C" EXTDLLIMPORTEXPOT XPRMdsfct *getXpressKalisModuleDefinition(void) {
    return extensions;
}

// Return the number of extensions in this module
extern "C" EXTDLLIMPORTEXPOT int getNumberOfXpressKalisExtensions(void) {
    return numNewFct;
}

XPRMnifct mm;                // Access to Mosel NI functions

// Retrieve the Mosel NI function table
extern "C" EXTDLLIMPORTEXPOT void setNiFCT(XPRMnifct nifct, u_iorp *interver,
                                           u_iorp *libver, XPRMdsointer **interf) {

```



```

    mm = nifct;
}

// Called upon module unloading
extern "C" EXTDLLIMPORTEXP void resetXpressKalisModule(void) {
    memoryManager.freeTemp();
}

```

The first two functions (`getXpressKalisModuleDefinition` and `getNumberOfXpressKalisExtensions`) pass the list of Mosel subroutines and their number from the user extension to Xpress-Kalis. The function `setNiFCT` retrieves the Mosel NI function table into the extension. The last function (`resetXpressKalisModule`) is called when the Xpress-Kalis module (and hence all extensions loaded by it) is unloaded. It is used in our example to free the memory allocated by our extension. **NB:** Modules are unloaded at the termination of Mosel or by an explicit module unloading command, and generally not after every single model run.

Note on memory management: every user extension needs to manage the memory it allocates. In our extension examples we use the memory manager defined in the include file `MemoryMgmt.h` that comes with the set of examples of the extensions package. This memory manager is provided in terms of an example, it is not a part of Xpress-Kalis and users may choose to implement other memory management procedures for their extensions. The declaration of the memory manager used in our implementation looks as follows:

```

#include "MemoryMgmt.h"
static MemMan memoryManager;

```

7.3.3 Implementation of the Mosel function

The following code extract is the C implementation of the Mosel function `diffEZ`: it exchanges information with Mosel through the Mosel NI and uses the Kalis library to create a new 'CP constraint' object.

```

int dez_diffEZConstraint(XPRMcontext ctx, void *libctx)
{
    s_mkctx *mkctx;
    Kobject *var1, *var2;

    try {
        mkctx = (s_mkctx *)libctx;

        // Obtain the arguments from Mosel.
        var1 = (Kobject *)XPRM_POP_REF(ctx);
        var2 = (Kobject *)XPRM_POP_REF(ctx);

        if((var1 == NULL) || (var2 == NULL))
        {
            printf("DiffEZ: Undefined cpvar in constraint creation.\n");
            return XPRM_RT_ERROR;
        }
        else
        {
            // Build the new Kobject representing the constraint
            Kobject *dezctr = new Kobject;

            // Set its class (as a constraint)
            dezctr->cls = C_KConstraint;

            // Set its value
            dezctr->value.any = new DiffEZConstraint(*var1->value.pKIntVar,
                                                    *var2->value.pKIntVar, ctx);

            // Add "dezctr" to the memory manager
            memoryManager.newObject(CCO_KConstraint, dezctr->value.pKConstraint);
            memoryManager.newObject(CCO_Kobject, dezctr);

            // Return result to Mosel
            XPRM_PUSH_REF(ctx, dezctr);
        }
    }
}

```

```

        // Everything went alright: return 'Xpress-Mosel RunTime ok'.
        return XPRM_RT_OK;
    }
} catch(ArtelysException &e) {
    printf("DiffEZ: Artelys Exception occurred during constraint creation.\n");
    mkctx->saved_ctx=NULL;
    return XPRM_RT_ERROR;
} catch (...) {
    printf("DiffEZ: Unknown exception occurred during constraint creation.\n");
    mkctx->saved_ctx=NULL;
    return XPRM_RT_ERROR;
}

// Everything went alright: return 'Xpress-Mosel RunTime ok'.
return XPRM_RT_OK;
}

```

Syntax: The C function called by Mosel as the implementation of the subroutine `diffez` has a fixed format, specified by the Mosel Native Interface (NI). The first argument is always the Mosel execution context and the second is an (optional) pointer to the library context, that is, for all Xpress-Kalis extensions, the execution context of the *kalis* module.

Subroutine arguments: The original arguments of the Mosel subroutine need to be retrieved in the order of their definition from the Mosel stack, using the macros `XPRM_POP_REF`, `XPRM_POP_INT`, `XPRM_POP_REAL`, or `XPRM_POP_STRING` provided by the Mosel NI for this purpose. Any objects other than Mosel's basic types are passed by reference, meaning that we need to use `XPRM_POP_REF` to access the stack for Xpress-Kalis types or any kind of structured data. Please note that always *all* arguments must be taken from the stack even if they might not be required in some specific cases.

Xpress-Kalis types: In the present example both arguments have the type `cpvar` in the Mosel language. Within Kalis this type corresponds to the class `KIntVar`. However, all Xpress-Kalis types are passed between Mosel and the extension as objects of the class `Kobject` and the two CP decision variables therefore need to be retrieved as such and are later converted to the desired type. The same applies to the new constraint created subsequently: the constraint is declared as a `Kobject`, its actual class is specified in the attribute `cls`, and the constraint definition is saved into the `value` of the object. The table 7.1 lists the Mosel types defined by *kalis*, the corresponding class in the Artelys Kalis library, and the type of the `Kobject` in Xpress-Kalis extensions.

Table 7.1: Correspondence between Mosel types and library classes

Mosel type	Description	Kalis class	Kobject type
<code>cpvar</code>	Discrete (finite-domain) variable	<code>KIntVar</code>	<code>C_KIntVar</code>
<code>cpfloatvar</code>	Continuous variable	<code>KFloatVar</code>	<code>C_KFloatVar</code>
<code>cpctr</code>	Constraint	<code>KConstraint</code>	<code>C_KConstraint</code>
<code>cpbranching</code>	Branching scheme	<code>KBranchingScheme</code>	<code>C_KbranchingScheme</code>
<code>cpvarlist</code>	List of discrete variables	<code>KIntVarArray</code>	<code>C_KintVarArray</code>
<code>cpdisjlist</code>	List of disjunctions	<code>KDisjunctionArray</code>	<code>C_KdisjunctionArray</code>
<code>cptask</code>	Task object	<code>KTask</code>	<code>C_KTask</code>
<code>cptasklist</code>	List of tasks	<code>KTaskArray</code>	<code>C_KTaskArray</code>
<code>cpresource</code>	Resource object	<code>KResource</code>	<code>C_KResource</code>

Return value and result of execution: The Mosel subroutine we wish to implement is a function, that means we need to pass its return value back to Mosel, via the Mosel stack. Since we return an Xpress-Kalis object (a new constraint) we need to use the stack access macro `XPRM_PUSH_REF`.

Error handling on the C level is implemented via the result value of the C function (codes `XPRM_RT_OK` or `XPRM_RT_ERROR`).

7.3.4 The Kalis user constraint class

A user-defined constraint for the Kalis library is derived from the abstract class `KUserConstraint`

(most general case) or defined as a specialization of one of the existing constraint classes.— Here we show the more general case, *i.e.*, departing from `KUserConstraint`. Besides a constructor and a destructor, every constraint class needs to implement the methods

- `awake` (what to do on first activation of the constraint),
- `propagate` (the implementation of the constraint propagation algorithm properly speaking), and
- `askIfEntailed` (employed within composite constraint relations).

It is also recommended to provide a `print` method. Optionally, specific propagation behavior can be implemented by defining one or several of the awake event methods:

- `awakeOnInf` (modification to lower bound),
- `awakeOnSup` (modification to upper bound),
- `awakeOnInst` (instantiation),
- `awakeOnRem` (removal of a value),
- `awakeOnVar` (used with Boolean connectors).

If any of these methods are not defined, the `propagate` method will be called in their place. For complicated constraint relations the specific propagation events may all be redefined to call `constAwake`. This signals to the solver that the constraint propagation algorithm for the particular constraint should be triggered only once all other propagations have been carried out at a node, thus avoiding repeated, unnecessarily time-consuming constraint evaluations.

For the DIFFEZ constraint we have the following class definition (in file `diffez.h`):

```
class DiffEZConstraint: public KUserConstraint
{
protected:
    XPRMcontext moselctx;

public:
    DiffEZConstraint(KIntVar &x, KIntVar &y, XPRMcontext ctx);
    virtual ~DiffEZConstraint() {};

    virtual void awake(void);
    virtual void awakeOnInst(KIntVar &var);
    virtual void propagate(void);
    virtual void print();
    virtual void print(void*ctx, PrintFunctionPtr*pf);
    virtual int askIfEntailed(void);
};
```

Let us now take a look at the implementation in detail:

Constructor: every class needs to implement at least one constructor. The constraint class constructor calls the super constructor 'KUserConstraint' with the variables of the constraint as the argument (one or two single variables as is the case in this example, or an array of variables `KIntVarArray`). The constructor of our constraint defines the constraint name to be used when printing the constraint and saves the Mosel context (not used in our example).

```
DiffEZConstraint::DiffEZConstraint(KIntVar &v1, KIntVar &v2,
XPRMcontext ctx): KUserConstraint(v1,v2) {
    char buf[80];
    snprintf(buf,80,"DiffEZ(%s,%s)", v1.getName(), v2.getName());
    setName(buf); // Set the constraint name

    moselctx = ctx; // Save Mosel context
}
```

Constraint propagation: On activation of a DIFFEZ constraint (*awake*) we simply call its propagation algorithm. The constraint propagation algorithm itself is implemented by the method *propagate*: if one of the two variables in the constraint is instantiated to a value different from 0, this value is removed from the domain of the second variable in the constraint. We have also implemented a specific handling of the event 'a variable has been instantiated' for this constraint class (method *awakeOnInst*): if the instantiated variable has a value different from 0 then this value is removed from the domain of the other variable in the constraint. The entailment test (method *askIfEntailed*) checks the different cases that might occur: only if both variables are instantiated we can decide whether the constraint is violated or not, in all other cases the return value is the status *CUNKNOWN*, or 'undecidable'.

```
// **** Initial propagation of the constraint ****
void DiffEZConstraint::awake() {
    propagate();
}

// **** Constraint propagation ****
void DiffEZConstraint::propagate() {
    if(_vars[0].getIsInstantiated() && _vars[0].getValue() != 0 &&
        _vars[1].canBeInstantiatedTo(_vars[0].getValue()))
        _vars[1].remVal(_vars[0].getValue());
    if(_vars[1].getIsInstantiated() && _vars[1].getValue() != 0 &&
        _vars[0].canBeInstantiatedTo(_vars[1].getValue()))
        _vars[0].remVal(_vars[1].getValue());
}

// **** Entailment test (used by composite constraints, e.g., 'implies') ****
int DiffEZConstraint::askIfEntailed() {
    if (_vars[0].getIsInstantiated() && _vars[1].getIsInstantiated() ) {
        if ( (_vars[0].getValue() != 0) &&
            (_vars[0].getValue() == _vars[1].getValue()) )
        {
            // the constraint is violated
            return CFALSE;
        }
        else { // the constraint is satisfied
            return CTRUE;
        }
    }
    else if ( (_vars[0].getIsInstantiated() && _vars[0].getValue() == 0) ||
        (_vars[1].getIsInstantiated() && _vars[1].getValue() == 0) ) {
        return CTRUE;
    }
    else {
        // Don't know yet if the constraint is definitely violated or verified
        return CUNKNOWN;
    }
}

// **** The variable 'var' has been instantiated to var.getValue() ****
void DiffEZConstraint::awakeOnInst(KIntVar &var) {
    if ( var.isEqualTo(_vars[0]) && var.getValue() != 0 ) {
        _vars[1].remVal(var.getValue());
    }
    else if ( var.isEqualTo(_vars[1]) && var.getValue() != 0 ) {
        _vars[0].remVal(var.getValue());
    }
}
```

Printing: the following two (optional) printing functions are defined by our constraint class. The second version is used, for instance, for displaying the constraint within IVE.

```
void DiffEZConstraint::print() {
    printf("%s", getName());
}

void DiffEZConstraint::print(void*ctx, PrintFunctionPtr*pfp)
{
    char buf[80];
    snprintf(buf, 80, "%s", getName());
    (*pfp)(ctx, buf);
}
```

7.3.5 Improving the constraint propagation algorithm

Sometimes it may be helpful with the implementation of constraint propagation algorithms to be able to save additional information that gets updated when the solver backtracks to an earlier node. This functionality is provided by Kalis through the concept of *annotations*, i.e., backtrackable arrays of integers. In this section we show how to use an annotation for implementing a *constraint status marker*.

Once a variable in the DIFFEZ constraint has been fixed and—given the instantiation value is different from zero—the corresponding value has been removed from the other variable's domain no further deductions are possible. In the interest of an efficient implementation it might therefore appear worthwhile to define a status flag 'do not evaluate this constraint again' applicable to the current node and the subtree under this node. We refer to this deactivation of the constraint as *freezing* the constraint. Once the search backtracks beyond the node where the relation has been frozen, the constraint becomes reactivated and is fully evaluated again.

The new constraint class definition looks as follows (notice the new attribute `propStatus` and the additional methods `freeze` and `isFrozen`).

```
class DiffEZConstraint: public KUserConstraint
{
protected:
    XPRMcontext moselctx;
    KIntSetIntAnnotation *propStatus; // 0: constraint is frozen, 1: propagate

public:
    DiffEZConstraint(KIntVar &x, KIntVar &y, KProblem *problem,
                    XPRMcontext ctx);
    virtual ~DiffEZConstraint();
    virtual void awake(void);
    virtual void awakeOnInst(KIntVar &var);
    virtual void propagate(void);
    virtual void print();
    virtual void print(void*ctx, PrintFunctionPtr*pf);
    virtual int askIfEntailed(void);

protected:
    void freeze(void);
    bool isFrozen(void);
};
```

Changes to the implementation: The constraint constructor initializes the problem status flag as an annotation with a single entry (index value 1) and sets its initial value to 1. Annotations being created within a problem, we pass the Kalis problem as an additional argument to the constructor. We now also need to define a destructor to delete the problem status marker.

```
DiffEZConstraint::DiffEZConstraint(KIntVar &v1, KIntVar &v2, KProblem *problem,
XPRMcontext ctx): KUserConstraint(v1,v2) {
    char buf[80];
    snprintf(buf,80,"DiffEZ(%s,%s)", v1.getName(), v2.getName());
    setName(buf); // Set the constraint name

    moselctx = ctx; // Save Mosel context

    propStatus = new KIntSetIntAnnotation(problem, getName(), 1, 1, 1);
}

DiffEZConstraint::~DiffEZConstraint()
{
    delete propStatus;
};
```

Point of view propagation algorithms, there are no changes to the 'awake' and 'askIfEntailed' methods. We show below the definition of the access methods `freeze` and `isFrozen` and the modified implementation of the propagation algorithm: if the constraint is frozen or if a variable is fixed to 0 no further checks or propagation are carried out, otherwise, if one variable is fixed to a non-zero value we remove this value from the other variable's domain (just as

before) and freeze the constraint (no further deductions are possible and hence, there is no need to evaluate this constraint again in this subtree). Similar modifications, and in particular the test whether the constraint is frozen, can be made for the `awakeOnInst` method.

```
void DiffEZConstraint::freeze() {
    propStatus->setIntAnnotation(1,0);
}

bool DiffEZConstraint::isFrozen() {
    return (propStatus->getIntAnnotation(1)==0);
}

void DiffEZConstraint::propagate() {
    if (isFrozen()) return;
    /* Case 0: a variable has been fixed to 0 */
    if((_vars[0].getIsInstantiated() && _vars[0].getValue()==0) ||
        (_vars[1].getIsInstantiated() && _vars[1].getValue()==0))
    {
        freeze();
        return;
    }
    /* Case 1: variable 1 has been fixed to a single value */
    if(_vars[0].getIsInstantiated() && _vars[0].getValue()!=0 &&
        _vars[1].canBeInstantiatedTo(_vars[0].getValue()))
    {
        _vars[1].remVal(_vars[0].getValue());
        freeze();
    }
    else /* Case 2: variable 2 has been fixed to a single value */
    if(_vars[1].getIsInstantiated() && _vars[1].getValue()!=0 &&
        _vars[0].canBeInstantiatedTo(_vars[1].getValue()))
    {
        _vars[0].remVal(_vars[1].getValue());
        freeze();
    }
}
```

The act of ‘freezing’ the DIFFEZ constraint reduces the number of full evaluations of the `propagate` method from 46 to 34 in our small example—on a larger scale this is likely to result in a non-negligible gain of speed.

Chapter 8

Writing a user branching scheme

The problem described in this chapter is taken from Section 9.1 ‘Wagon load balancing’ of the book ‘Applications of optimization with Xpress-MP’

A number of railway wagons with a fixed carrying capacity has been reserved to transport a load of boxes. The weight of the boxes in quintals is given in the following table. How shall the boxes be assigned to the wagons in order to keep to the limits on the maximum carrying capacity and to minimize the heaviest wagon load? **NB:** we work here with a larger problem instance than the one in the original problem description.

Table 8.1: Weight of boxes

Box	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Weight	34	6	8	17	16	5	13	21	25	31	14	13	33	9	25	25
Box	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Weight	35	6	8	12	16	5	13	21	27	30	11	13	33	9	28	26
Box	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Weight	14	6	8	18	16	5	3	41	5	31	14	23	32	7	12	27
Box	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
Weight	15	6	8	12	16	5	23	21	27	3	11	43	31	7	12	29

A heuristic, not necessarily optimal solution to this problem can be obtained with the following procedure: until all boxes are distributed to the wagons we choose in turn the heaviest unassigned box and put it onto the wagon with the least load. This heuristic is known as the *Longest Processing Time (LPT)* heuristic.

8.1 Model formulation

Let $BOXES = \{1, \dots, NB\}$ be the set of boxes, $WAGONS = \{1, \dots, NW\}$ the set of wagons, $WEIGHT_b$ the weight of box b and $WMAX$ the maximum carrying load of a wagon. The assignment of the boxes to the wagons is represented by discrete variables $load_b$ that take the value w of the wagon that the box is assigned to. We also introduce auxiliary variables $weight_{bw}$ that take the value $WEIGHT_b$ if box b is assigned to wagon w and 0 otherwise. The two sets of variables are connected by the following logical relations.

$$\forall b \in BOXES, \forall w \in WAGONS : load_b = w \Leftrightarrow weight_{bw} = WEIGHT_b$$

In this problem we wish to minimize the maximum load of the wagons. Such an objective is sometimes referred to as *minimax* objective. We define a non-negative variable $maxweight$ to represent the maximum weight over all the wagon loads. The objective function consists of minimizing $maxweight$. The following constraints are established to set $maxweight$ as the upper bound on every wagon load, alternatively we might use a ‘maximum’ constraint.

$$\forall w \in WAGONS : \sum_{b \in BOXES} weight_{bw} \leq maxweight$$

By proceeding this way, in the optimal solution the minimization will force *maxweight* to take the value that corresponds to the weight of the heaviest wagon load.

8.2 Implementation: Mosel model

The following Mosel model shows how the idea of the LPT heuristic can be used to define a user branching strategy: the variable selection order is fixed based on the weight of the boxes (in decreasing order) and a user-defined value selection function chooses the wagon with the least load for a given box.

```

model "D-1 Wagon load balancing (CP)"
uses "kalis", "mmsystem"

forward function valchoice(x: cvar): integer

declarations
  NB = 64
  NW = 5
  BOXES = 1..NB
  WAGONS = 1..NW
  WEIGHT: array(BOXES) of integer
  WMAX: integer
  BSORT: array(BOXES) of integer
  load: array(BOXES) of cvar
  weight: array(BOXES,WAGONS) of cvar
  maxweight: cvar
  vlist: cvarlist
  minW: array(WAGONS) of integer
end-declarations

initializations from 'dlwagon2.dat'
  WEIGHT WMAX
end-initializations

! Defining the variables
forall(b in BOXES) do
  setdomain(load(b), WAGONS)
  forall(w in WAGONS) do
    setdomain(weight(b,w), {0,WEIGHT(b)})
    equiv(load(b)=w, weight(b,w)=WEIGHT(b))
  end-do
end-do

! Setting bounds on maximum weight
setdomain(maxweight, ceil((sum(b in BOXES) WEIGHT(b))/NW), WMAX)
forall(w in WAGONS) sum(b in BOXES) weight(b,w) <= maxweight

! Definition of search (fixed variable selection, user value selection)
qsort(SYS_DOWN, WEIGHT, BSORT)
forall(b in BOXES) vlist+= load(BSORT(b))
cp_set_branching(assign_var(KALIS_INPUT_ORDER, "valchoice", vlist))

! Problem solving
if cp_minimize(maxweight) then
  writeln("Solution: Max weight: ", getval(maxweight))
end-if

! *****
! *** Value choice: choose the wagon with the least load
function valchoice(x: cvar): integer

  if (getlb(x)=getub(x)) then
    returned:=getlb(x)
  else
    forall(w in WAGONS) minW(w):= sum(b in BOXES | is_fixed(load(b)) and
      getval(load(b))=w) WEIGHT(b)
    ct:=getlb(x)
  end-if
end-function

```



```

        returned:= ct
        minl:= minW(ct)
        while(ct<getub(x)) do          ! Determine wagon with least load in the domain
            nextval:= getnext(x,ct)
            ct:= nextval
            if minW(ct)<minl then
                returned:= ct
                minl:= minW(ct)
            end-if
        end-do

    end-if
end-function

end-model

```

The following model version uses the same branching strategy as the one printed above. This time the branching strategy is not defined directly in the Mosel model, we use instead the 'LPT' branching scheme defined in the user extension `lptbranch` that is presented in the next section.

```

model "D-1 Wagon load balancing (using extension)"
uses "kalis"

declarations
    NB = 64
    NW = 5
    BOXES = 1..NB                ! Set of boxes
    WAGONS = 1..NW               ! Set of wagons

    WEIGHT: array(BOXES) of integer ! Box weights
    WMAX: integer                ! Weight limit per wagon

    load: array(BOXES) of cpvar   ! Wagon the box is loaded on
    weight: array(BOXES,WAGONS) of cpvar ! Weight of box loaded on wagon
    maxweight: cpvar              ! Weight of the heaviest wagon load
end-declarations

initializations from 'dlwagon2.dat'
    WEIGHT WMAX
end-initializations

! Defining the variables
forall(b in BOXES) do
    setdomain(load(b), WAGONS)          ! load(b)=w iff box b on wagon w
    forall(w in WAGONS) do
        setdomain(weight(b,w), {0,WEIGHT(b)})
        equiv(load(b)=w, weight(b,w)=WEIGHT(b))
    end-do
end-do

! Setting bounds on maximum weight
setdomain(maxweight, ceil((sum(b in BOXES) WEIGHT(b))/NW), WMAX)
forall(w in WAGONS) sum(b in BOXES) weight(b,w) <= maxweight

! Definition of search (branching scheme 'LPT' defined in user extension)
cp_set_branching(LPT(load, WEIGHT))

! Problem solving
if cp_minimize(maxweight) then
    writeln("Solution: Max weight: ", getval(maxweight))
end-if

end-model

```

This search strategy immediately finds a first solution of value 225, followed by a second solution of value 224 that is proven optimal within 8354 nodes (run time about 1 second on a standard PC).

Table 8.2: Optimal distribution of boxes onto wagons

Wagon	Weight	Boxes
1	224	8 9 10 11 18 21 30 34 48 58 59 60 63
2	224	14 15 22 24 28 37 40 41 42 43 52 54 57
3	223	3 4 16 17 23 26 33 46 47 51 55 61
4	224	1 2 5 7 27 32 35 39 44 45 53 62 64
5	224	6 12 13 19 20 25 29 31 36 38 49 50 56

8.3 Implementation: user extension

The structure of a user extension defining a branching scheme is very similar to what we have seen in the previous chapter for the definition of a new constraint relation. In the first place, we shall discuss the case that corresponds directly to the Mosel implementation of the user search strategy shown above, namely the implementation of a user value selection strategy that is employed with a predefined branching scheme ('assignVar') and a fixed variable ordering. At the end of this chapter we also explain how to proceed in the more general case, that is, how to implement a complete new branching scheme with user value and variable selection strategies defined on the library level.

Here is once more the list of the major components of a user extension:

1. the declaration of the new subroutine(s) for the Mosel language,
2. the definition of the four Xpress-Kalis interface functions,
3. the implementation of the C function(s) associated with the entries in the list of Mosel subroutines,
4. the implementation of the C++ class(es) and any related methods as required by the extension mechanisms of the Artelys Kalis library.

For the definition of the LPT branching scheme we have a single entry in the table of subroutines and correspondingly the definition of one C function implementing the subroutine. The value selection is implemented by a class derived from the `KValueSelector` class of the Kalis library. For a complete branching scheme we also need to implement new classes inheriting from `KBranchingScheme` and `KVariableSelector`.

8.3.1 List of Mosel subroutines

The declaration of the prototype of the new subroutine for the Mosel language is very similar to what we have seen in Section 7.3.1 for the constraint function. Again, we have just a single entry in the list of subroutines.

```
static const int numNewFct = 1; // No. of new functions within this extension

static XPRMdsOfct extensions[numNewFct] =
{
    {
        "LPT", // name of the function within Mosel
        10001, // code (unique value within this extension)
        XPRM_TYP_EXTN, // return type (here: CP branching, an "external type")
        2, // number of arguments
        "cpbranching:A.|cpvar|A.i", // signature (external type names in arguments
        // are delimited by "|")
        lpt_createBranching // function called for branching scheme creation
    }
};
```

The signature of the Mosel subroutine `LPT` indicates that this is a function returning a new CP branching scheme (type `cpbranching`) with an array of finite domain variables as its first

argument and an array of integers as its second argument. The index sets of the arrays are not specified, this means that any type and number of indexing sets are accepted by Mosel and we need to test in the implementation of the C function `lpt_createBranching` whether the two arrays have the same indices.

8.3.2 Xpress-Kalis interface functions

The definition of the four Xpress-Kalis interface functions is exactly the same as what we have seen in Section 7.3.2.

8.3.3 Implementation of the Mosel function

The implementation of the Mosel function does some more work than in the previous example of constraint definition:

1. We need to make sure that the two arrays passed into the subroutine have the same index sets (we use the following tests: check whether the arrays have the same number of elements, the same number of dimensions, and the same index sets).
2. The decision variables must be sorted in decreasing order of the weight values.
3. The decision variable and weight arrays need to be transformed into one-dimensional arrays to conform with the format expected by the Kalis library for the branching variables.

```
int lpt_createBranching(XPRMcontext ctx,void *libctx)
{
    s_mkctx *mkctx;
    XPRMarray xArray, DArray;

    try {
        mkctx=(s_mkctx *)libctx;

        // Obtain the arguments from Mosel.
        xArray = (XPRMarray)XPRM_POP_REF(ctx);
        DArray = (XPRMarray)XPRM_POP_REF(ctx);

        // Obtain and compare array sizes and dimensions
        if((xArray == NULL) || (DArray == NULL))
        {
            mm->dispmsg(ctx, "LPTbranch: Empty array in argument.\n");
            return XPRM_RT_ERROR;
        }
        else if(mm->getarrsize(xArray)!=mm->getarrsize(DArray))
        {
            mm->dispmsg(ctx,"LPTbranch: Arrays have different sizes.\n");
            return XPRM_RT_ERROR;
        }
        else if(mm->getarrdim(xArray)!=mm->getarrdim(DArray))
        {
            mm->dispmsg(ctx,"LPTbranch: Arrays have different number of dimensions.\n");
            return XPRM_RT_ERROR;
        }
        else
        {
            int aDim = mm->getarrdim(DArray);
            XPRMset *setsx = new XPRMset[aDim];
            XPRMset *setsD = new XPRMset[aDim];
            mm->getarrsets(xArray,setsx);
            mm->getarrsets(DArray,setsD);
            for(int i=0;i<aDim;i++)
                if(setsx[i]!=setsD[i])
                {
                    mm->dispmsg(ctx,"LPTbranch: Arrays have different index sets.\n");
                    delete[] setsx;
                    delete[] setsD;
                    return XPRM_RT_ERROR;
                }
            delete[] setsx;
        }
    }
```

```

delete[] setsD;

int aDim = mm->getarrdim(DArray);
int *indices = new int[aDim];
int aSize = mm->getarrsize(DArray);
XPRMalltypes value;

// Copy all variables and weights into a one-dimensional array
LPTData *toSort = new LPTData[aSize];
int ct = 0;
mm->getfirstarrtruentry(xArray, indices);
do {
    (void)mm->getarrval(xArray, indices, &value);
    KIntVar var = *((KObject *) (value.ref))->value.pKIntVar;
    toSort[ct].var=var;
    mm->getarrval(DArray, indices, &value);
    toSort[ct].value=value.integer;
    ct++;
} while(!mm->getnextarrtruentry(xArray, indices));

// Sort the variable/value array
qsort(toSort, aSize, sizeof(LPTData), cmplpt);

// Copy sorted data into flat arrays
int *DValues = new int[aSize];
KIntVarArray *vars = new KIntVarArray;
for(int i=0; i<aSize;i++)
{
    DValues[i] = toSort[i].value;
    *vars+=toSort[i].var;
}

// Build the new KObject representing the branching scheme
Kobject *lptbranch = new Kobject;

// Set its class (as a branching scheme)
lptbranch->cls = C_KBranchingScheme;

// Set the variable and value selection strategies
KInputOrder *varSel = new KInputOrder();
LPTValSelector *valSel = new LPTValSelector(*vars, DValues);
lptbranch->value.any = new KAssignVar(*varSel, *valSel, *vars);

// Add new objects to the memory manager
memoryManager.newObject(CCO_KBranchingScheme, lptbranch->value.any);
memoryManager.newObject(CCO_KIntVarArray, vars);
memoryManager.newObject(CCO_Kobject, lptbranch);
memoryManager.newObject(CCO_KVariableSelector, varSel);
memoryManager.newObject(CCO_KValueSelector, valSel);
memoryManager.newObject(CCO_IntArray, DValues);
memoryManager.newObject(CCO_IntArray, indices);

delete[] toSort;

// Return result to Mosel
XPRM_PUSH_REF(ctx, lptbranch);

// Everything went alright: return "Xpress-Mosel RunTime ok".
return XPRM_RT_OK;
}
} catch(ArtelysException &e) {
    mm->dispmsg(ctx, "LPTbranch: Artelys Exception occured during constraint creation.\n");
    mkctx->saved_ctx=NULL;
    return XPRM_RT_ERROR;
} catch (...) {
    mm->dispmsg(ctx, "LPTbranch: Unknown exception occured during constraint creation.\n");
    mkctx->saved_ctx=NULL;
    return XPRM_RT_ERROR;
}

// Everything went alright: return "Xpress-Mosel RunTime ok".
return XPRM_RT_OK;
}

```

Syntax: The C function called by Mosel as the implementation of the subroutine `LPT` has a fixed format, specified by the Mosel Native Interface (NI). The first argument is always the Mosel execution context and the second is an (optional) pointer to the library context, that is, for all Xpress-Kalis extensions, the execution context of the *kalis* module.

Subroutine arguments: The original arguments of the Mosel subroutine need to be retrieved in the order of their definition from the Mosel stack. Here we use the macro `XPRM_POP_REF` (stack access for Xpress-Kalis types or any kind of structured data) since we are working with arrays. The reader is reminded that always *all* arguments must be taken from the stack even if some arguments are perhaps not required in a specific case.

Xpress-Kalis types: In this example both arguments have the type `array` in the Mosel language. The `array of real` is retrieved using standard Mosel functionality. The `array of cpvar` needs to be treated differently. Within Kalis the type `cpvar` corresponds to the class `KIntVar`. However, all Xpress-Kalis types are passed between Mosel and the extension as objects of the class `Kobject` and the decision variables therefore need to be retrieved as such and are then converted to `KIntVar`. The same remark applies to the new branching scheme `lptbranch` created by this function: it is declared as a `Kobject`, its actual class is specified in the attribute `cls`, and its definition is saved into the `value` of the object. For a list of the Mosel types defined by *kalis* and their correspondence on the library level please see the table 7.1 in the previous chapter.

Return value and result of execution: This library function implements a Mosel *function*, that means we need to pass the function return value back to Mosel, via the Mosel stack. Since we return an Xpress-Kalis object (a new branching scheme) we need to use the stack access macro `XPRM_PUSH_REF`.

Error handling on the C level is implemented via the result value of the C function (codes `XPRM_RT_OK` or `XPRM_RT_ERROR`).

For completeness' sake, here are the auxiliary data structure and comparison function that are used by the `qsort` function:

```
typedef struct {
    KIntVar var;
    int value;
} LPTData;

static int cmplpt(const void *a1,const void *a2)
{
    if(((LPTData *)a1)->value<((LPTData *)a2)->value) return 1;
    else
        if(((LPTData *)a1)->value>((LPTData *)a2)->value) return -1;
    else return 0;
}
```

8.3.4 The Kalis value selection strategy class

Our value selection class is derived from the Kalis class `KValueSelector`. Besides some standard methods (constructor, copy-constructor, destructor) every value selection class needs to define the method `selectNextValue` that specifies which value to choose next for branching on a given variable.

In summary, we have the following class definition for our LPT value selection strategy (in file `lptbranch.h`):

```
class LPTValSelector: public KValueSelector
{
protected:
    KIntVarArray *vars;
    int *Dvalues;           // Weight values
    int *Load;              // Load values
    int minVal,maxVal;      // Smallest / largest domain values for vars

public:
    LPTValSelector(KIntVarArray &vars, int *Dvals);
}
```

```

    LPTValSelector(const LPTValSelector &LPTValSelector);
    virtual ~LPTValSelector();
    virtual int selectNextValue(KIntVar* intVar);
    virtual KValueSelector* getCopyPtr() const;
};

```

Here follows the complete listing of the implementation of this class. Within the constructor method we determine the smallest and largest values occurring in all variables' domains and initialize the auxiliary array `Load` that is used by the implementation of the variable selection. The method `getCopyPtr` is used internally by the Kalis library for memory management purposes.

```

// Constructor
LPTValSelector::LPTValSelector(KIntVarArray &varArray, int *DVals)
{
    int i;
    vars = new KIntVarArray(varArray);
    memoryManager.newObject(CCO_KIntVarArray, vars);
    Dvalues = new int[vars->getNumberOfElements()];
    for(i=0; i<vars->getNumberOfElements(); i++) Dvalues[i] = DVals[i];

    minVal = vars->getElt(0)->getInf();
    maxVal = vars->getElt(0)->getSup();
    for(i=1; i<vars->getNumberOfElements(); i++)
    {
        if(vars->getElt(i)->getInf()<minVal) minVal=vars->getElt(i)->getInf();
        if(vars->getElt(i)->getSup()>maxVal) maxVal=vars->getElt(i)->getSup();
    }
    Load = new int[maxVal-minVal+1];
    memset(Load, 0, (maxVal-minVal+1)*sizeof(int));
    memoryManager.newObject(CCO_IntArray, Load);
    memoryManager.newObject(CCO_IntArray, Dvalues);
}

LPTValSelector::LPTValSelector(const LPTValSelector &LPTValSelectorToCopy)
{
    vars = LPTValSelectorToCopy.vars;
    Dvalues = LPTValSelectorToCopy.Dvalues;
    minVal = LPTValSelectorToCopy.minVal;
    maxVal = LPTValSelectorToCopy.maxVal;
    Load = new int[maxVal-minVal+1];
    memoryManager.newObject(CCO_IntArray, Load);
    memcpy(Load, LPTValSelectorToCopy.Load, sizeof(LPTValSelectorToCopy.Load));
}

LPTValSelector::~~LPTValSelector()
{
    // nothing to be done
}

KValueSelector* LPTValSelector::getCopyPtr() const
{
    return new LPTValSelector(*this);
}

```

The value selection strategy properly speaking is implemented by the method `selectNextValue`. If the variable has just a single value left in its domain we return this value. Otherwise, we calculate the fixed load for each wagon and return the value from the domain of our branching variable that is associated with the least load.

```

int LPTValSelector::selectNextValue(KIntVar *aVar)
{
    // Return value in the variable's domain associated with the least load
    int i=0, minl;
    int val;

    if (aVar->getInf() == aVar->getSup()) val = aVar->getInf();
    else
    {
        memset(Load, 0, (maxVal-minVal+1)*sizeof(int));

        // Calculate load for all feasible values

```

```

for(i=0; i<vars->getNumberOfElements(); i++)
    if(vars->getElt(i)->getIsInstantiated())
        Load[(int)round(vars->getElt(i)->getValue()) - minVal] += Dvalues[i];

// Return value of least load
i = aVar->getInf();
minl = Load[i-minVal];
val = i;
int next = i;
while(next<aVar->getSup())
{
    aVar->getNextDomainValue(next);
    i = next;
    if(Load[i-minVal]<minl)
    {
        val = i;
        minl = Load[i-minVal];
    }
}
return val;
}

```

8.4 Implementing a complete branching scheme

The previous section defines a new branching scheme for the Mosel language that is formed (on the library level) by a predefined branching scheme used with a predefined variable selection strategy and a user-implemented value selection strategy. We now show how to implement a complete branching scheme in an XPress-Kalis extension. We might wish, for instance, to define a second version of the LPT branching scheme with an additional argument (an integer) denoting the maximum number of branches from each node. This new version can be added to the list of subroutines of the extension library `lptbranch` or be implemented as a separate file, resulting in a second extension library, say `lptbranch2`. In both cases, as always with Mosel modules, we may use the same name for the subroutine in the Mosel language, that is, define an overloaded version. The *rules for overloading Mosel subroutines* the following: (1) the overloaded versions must be different from each other in at least one (type of) argument, and (2) all subroutines of the same name must be either all procedures or all functions.

From the Mosel point of view only minor modifications are required for this new version: the signature in the entry in the list subroutines includes the third argument:

```
"cpbranching:A.|cpvar|A.ii"
```

the value of which needs to be retrieved from the stack in the implementation of the Mosel function. Other modifications to the function `lpt_createBranching` relate to the new classes we are about to define:

```
// Set the variable and value selection strategies
lptbranch->value.any = new LPTAssignVar(vars, DValues, maxb);
```

The creation of the variable and value selection strategy objects have been moved into the constructor of the `LPTAssignVar` branching scheme, thus saving on the creation of one copy of each of these.

8.4.1 The Kalis variable selection strategy class

In the present example there really is no necessity of implementing our own variable selection strategy. Nevertheless, for completeness' sake we show re-implementation of the 'INPUT_ORDER' selection criterion.

There is no need to store any additional information with this variable selector, the class definition therefore simply consists of the minimum set of methods required of variable selectors:

```

class LPTVarSelector: public KVariableSelector
{
public:
    LPTVarSelector(void);
    LPTVarSelector(const LPTVarSelector &LPTVarSelector);
    virtual ~LPTVarSelector();
    virtual KIntVar* selectNextVariable(KIntVarArray* intVarArray);
    virtual KVariableSelector* getCopyPtr() const;
};

```

Nothing specific needs to be done by the constructors and correspondingly the destructor, their definition is empty. The actual implementation of the strategy is in the method `selectNextVariable`: this method returns the first uninstantiated variable that is found in the array of variables to be enumerated, or `NULL` if none is found.

```

KIntVar* LPTVarSelector::selectNextVariable(KIntVarArray* vars)
{
    // Return first uninstantiated variable in sorted order
    int i=0;

    while ((i<vars->getNumberOfElements()) &&
           (vars->getElt(i)->getIsInstantiated())) i++;
    if(i<vars->getNumberOfElements())
        return vars->getElt(i);
    else
        return NULL;
}

```

8.4.2 The Kalis branching scheme class

The branching scheme shown below inherits from the class `KBranchingScheme`. Besides the constructors/destructors, a branching scheme needs to implement a list of specific methods:

- `selectNextBranchingObject`: selection of the object to branch on (here; a decision variable)
- `finishedBranching`: whether all branches from a node are done
- `getNextBranch`: determine how to form the next branch (here: select a value)
- `goDownBranch`: what to do on entering a branch
- `goUpBranch`: what to do when moving up out of a branch
- `freeAllocatedObjectsForBranching`: for memory management related to branching

This is the class definition for our 'AssignVar with limit on branches' scheme:

```

class LPTAssignVar: public KBranchingScheme
{
protected:
    KIntVarArray* vars;
    LPTVarSelector* varSelect;
    LPTValSelector* valueSelect;
    int maxBranch;

public:
    // Constructors
    LPTAssignVar(KIntVarArray* intVarArray, int *Dvals);
    LPTAssignVar(KIntVarArray* intVarArray, int *Dvals, int mb);
    // Copy constructor
    LPTAssignVar(const LPTAssignVar& LPTAssignVarToCopy);
    // Destructor
    virtual ~LPTAssignVar();

    // Methods
    virtual void* selectNextBranchingObject();
}

```



```

        virtual bool finishedBranching(void* branchingObject,
        void* branchingInformation, int currentBranchNumber);
        virtual void* getNextBranch(void* branchingObject,
        void* branchingInformation, int currentBranchNumber);
        virtual void goDownBranch(void* branchingObject,
        void* branchingInformation, int currentBranchNumber);
        virtual void goUpBranch(void* branchingObject,
        void* branchingInformation, int currentBranchNumber);
        virtual void freeAllocatedObjectsForBranching(void* branchingObject,
        void* branchingInformation);
        virtual KBranchingScheme* getCopyPtr() const;
};

```

The constructor creates and saves the variable and value selection strategies to be used within the branching scheme and the limit on the branch number specified when calling the branching scheme in the Mosel model.

```

LPTAssignVar::LPTAssignVar(KIntVarArray* intVarArray, int *DVals): KBranchingScheme()
{
    varSelect = new LPTVarSelector();
    valueSelect = new LPTValSelector(*intVarArray, DVals);
    memoryManager.newObject(CCO_KVariableSelector, varSelect);
    memoryManager.newObject(CCO_KValueSelector, valueSelect);
    vars = intVarArray;
    maxBranch = -1;
}

LPTAssignVar::LPTAssignVar(KIntVarArray* intVarArray, int *DVals, int mb):
    KBranchingScheme()
{
    varSelect = new LPTVarSelector();
    valueSelect = new LPTValSelector(*intVarArray, DVals);
    memoryManager.newObject(CCO_KVariableSelector, varSelect);
    memoryManager.newObject(CCO_KValueSelector, valueSelect);
    vars = intVarArray;
    maxBranch = mb;
}

LPTAssignVar::LPTAssignVar(const LPTAssignVar &LPTAssignVarToCopy)
{
    varSelect = LPTAssignVarToCopy.varSelect;
    valueSelect = LPTAssignVarToCopy.valueSelect;
    vars = LPTAssignVarToCopy.vars;
    maxBranch = LPTAssignVarToCopy.maxBranch;
}

LPTAssignVar::~~LPTAssignVar()
{
    // nothing to be done
}

```

The selection of the next branching object simply calls our variable selection strategy. If a limit on the number of branches from a node has been specified this value is checked by the finishedBranching method. The getNextBranch method calls our value selection strategy.

```

// Select next branching object
void* LPTAssignVar::selectNextBranchingObject() {
    return varSelect->selectNextVariable(vars);
}

// Indicate whether all branches from a node have been explored
// (branch numbering starts with 1)
bool LPTAssignVar::finishedBranching(void* branchingObject,
    void* branchingInformation, int currentBranchNumber)
{
    if (maxBranch > -1)
        return ( (currentBranchNumber >= maxBranch) ||
            ( ((KIntVar*) branchingObject)->getDomainSize() < 1) );
    else
        return ( ((KIntVar*) branchingObject)->getDomainSize() < 1);
}

```

```

// Getting information for next branch
void* LPTAssignVar::getNextBranch(void* branchingObject,
    void* branchingInformation, int currentBranchNumber)
{
    int nextValue = valueSelect->selectNextValue((KIntVar*) branchingObject);
    int *ret = new int[1];
    ret[0] = nextValue;
    memoryManager.newObject(CCO_IntArray, ret);
    return ret;
}

```

In the 'AssignVar' scheme a new branch is formed by fixing the branching variable to the chosen value (`goDownBranch`). When moving up out of the branch that has just been explored the branching value is removed from the variable's domain (`goUpBranch`). In this implementation nothing needs to be done to free branching objects since the only information saved (the branching value) is handled by our memory manager.

```

// Going down one level in the branching tree
void LPTAssignVar::goDownBranch(void* branchingObject,
    void* branchingInformation, int currentBranchNumber)
{
    ((KIntVar*) branchingObject)->instantiate((*((int*) branchingInformation)));
}

// Moving up one level in the branching tree
void LPTAssignVar::goUpBranch(void* branchingObject,
    void* branchingInformation, int currentBranchNumber)
{
    ((KIntVar*) branchingObject)->remVal((*((int*) branchingInformation)));
}

// Delete allocated objects
void LPTAssignVar::freeAllocatedObjectsForBranching(void* branchingObject,
    void* branchingInformation)
{
    // Handled by memory manager
}

KBranchingScheme* LPTAssignVar::getCopyPtr() const
{
    return new LPTAssignVar(*this);
}

```

Appendix

Appendix A

Trouble shooting

- **No license found:** to work with Kalis for Mosel, the Xpress-MP licensing system, and the Xpress-Kalis module must be installed. You need to copy the license file that you will receive from your software vendor into the Xpress-MP installation directory and set the environment variable `XPRESSDIR` to point to this directory.
- **The Xpress-Kalis module is not found:** if the file `kalis.dso` is not installed in the directory `dso` of the Mosel distribution, then the environment variable `MOSEL_DSO` must be defined with the location of this file.

Appendix B

Glossary of CP terms

Some terms commonly used in CP might require some explanation for readers with an Operations Research background. The following list is an extract from [Hei99].

Finite domain constraint problem/constraint satisfaction problem (CSP): defined by a finite set of variables taking values from *finite domains* and a (conjunctive) set of constraints on these variables. The objective may be either finding one solution (any or an optimal) or all solutions (consistent assignment of values to the variables so that all the constraints are satisfied simultaneously) for the given instance. The term *constraint network* is frequently employed to denote CP problems in allusion to the graphical representation as a hyper graph (constraint graph), where nodes represent variables, and constraints are (hyper) arcs linking several nodes. There is no standard problem representation in CP.

Model: a CP model specifies all variables, their domains and their declarative meaning and *conceptual constraints* imposed on them (as opposed to *actual constraints* that are used to implement the properties of the solution and the search process). In CP in general, a model preserves much problem-specific knowledge about variables and the relations between them. This allows the development and application of more efficient specialized solution strategies.

Variable: object that has a name and a domain (also referred to as *decision variable*).

Domain: the set of values (also: labels) a variable may take. In Xpress-Kalis, it may consist of discrete values, or intervals of integers. When solving CP problems active use of the domain concept is made. At any stage, the domain of a variable is the set of values that cannot be proved to be inconsistent (with the constraints on this variable) using the available consistency checking methods. Assigning or restricting domains is often interpreted as unary constraints on the corresponding variables.

Instantiation of a set of variables is an assignment of a value to each variable from its domain, also called *labeling* of each variable with a value from its domain.

Consistent instantiation of a constraint network is an instantiation of the variables such that the constraints between variables are satisfied, also called *admissible/satisfied instantiation*, *consistent assignment of values*, or *consistent labeling*. *Solution* is often used as a synonym for consistent instantiation, but may also denote the result after applying any (local/partial) consistency algorithm.

Constraint: a relation over a set of variables limiting the combination of values that these variables can take; constraints may also be interpreted as mappings from the domains of the variables onto the Boolean values *true* and *false*. A (conceptual) constraint can sometimes be implemented in different ways enforcing various levels of consistency (see below) with different computational overhead. So-called *global constraints* subsume a set of other constraints (for instance an 'all-different' relation on a set of variables replaces pair wise disequality constraints). Global constraints use specific propagation/consistency algorithms that render them more efficient than the set of constraints they replace.

Redundant constraints: a constraint is redundant with respect to a set of constraints, if it is satisfied when the set of constraints is satisfied. Although redundant constraints do not change the set of solutions (consistent instantiations) of a problem, in practice it may be useful to add

redundant constraints to the model formulation because they can help CP solution procedures, particularly by achieving more powerful constraint propagation.

System of constraints: a conjunctive set of constraints, usually built up *incrementally*.

Constraint solving: deciding the consistency or satisfiability of a system of constraints.

Solution methods: finite domain CP problems are usually solved by tree search methods (Branch-and-Bound for optimization, Branch-and-Prune for decision problems) that enumerate the possible values of the variables coupled with consistency algorithms. In tree search methods with consistency checking the local consistency algorithm is triggered by the propagation of the domain changes of the branching variable. For *optimization* usually a cost constraint is introduced that propagates to the variables. It is updated (in the case of minimization: bounded to be smaller than the solution value) each time a new solution is found.

Consistency techniques and constraint propagation: Consistency algorithms remove inconsistent values from the domains of variables. Informally speaking, a consistency algorithm is 'stronger' than another one if it reduces the domains further, *i.e.*, it establishes a higher level of consistency. In finite domain CP, typically local consistency algorithms are used. *Local* or *partial consistency* signifies that only subsets of the constraints of a system of constraints are simultaneously satisfied. A locally consistent (according to some notion of consistency, such as arc-consistency) constraint network can be obtained by propagating iteratively the effects of each constraint to all other constraints it is connected to through its variables until a stable state is reached. This process is referred to as *constraint propagation*. Propagation properties of constraints vary, *e.g.*, due to their implementation, or the types of variables used. Possible events triggering their evaluation may be variable instantiation, modification of domain bounds, removing of value(s) from a domain, *etc.*

Backtrack search augmented by constraint propagation:

```
while not solved and not infeasible
  check/establish (local) consistency
  if a dead end is detected
    then backtrack to the first open node
  else
    select a variable
    select a value for the variable
```

Search algorithms/strategies: The values for variables come out of an enumeration process. 'Intelligent' enumeration strategies adapted to special types of constraints and variables are a central issue in CP. The search is controlled by problem specific heuristics, strategies from Mathematical Programming or the expert's knowledge; fixing variables to trial values is possible. One can distinguish variable and value selection heuristics. Due to the way the backtracking mechanism works, usually depth-first search is used.

Constraint solver: (Also: *constraint engine*.) Distinction between exact and incomplete solvers. *Exact* solvers guarantee the satisfiability of the system of constraints at any stage of the computations, they usually work on rational numbers (trees of rationals and linear constraints). *Incomplete* solvers are designed for more complex domains such as integers where checking and maintaining consistency of the overall system is too expensive or not possible with presently known algorithms. These solvers work with simplified calculations establishing some sort of partial (local) consistency among constraints; usually simply stating constraints does not produce a solution, an enumeration phase (searching for solutions) is necessary.

Bibliography

- [FM63] H. Fisher and J. F. Muth. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225—251, Englewood Cliffs, N. J., 1963. Prentice Hall.
- [Hei99] S. Heipcke. Comparing Constraint Programming and Mathematical Programming Approaches to Discrete Optimisation. The Change Problem. *Journal of the Operational Research Society*, 50(6):581—595, 1999.

Index

Symbols

`+`, 10

`;`, 7

A

`abs`, 18, 26

`all-different`, 119

`all-different constraint`, 29

`all_different`, 18, 25

`and`, 35

`annotation`, 103

`array`

definition, 6

`assign_var`, 15, 56

`assignment problem`, 59

`AUTO_PROPAGATE`, 21

B

`backtrack search`, 120

`binary constraint`, 50

`binpacking`, 72

`bottleneck machine`, 29

`bottleneck problem`, 59

`bound`

default, 12

lower, 62

upper, 62

`Branch-and-Bound`, 120

`branching scheme`, 15, 35, 46, 51, 56, 92

`branching strategy`, 2, 46, 51

C

`callback`, 28, 58

`ceil`, 36

`comments`, 7

`compile model`, 7

`COMPUTATION_TIME`, 24

`condition`

loop, 11

`consistency`

local, 120

partial, 120

`consistency algorithm`, 119, 120

`consistent instantiation`, 2

`constraint`, 2, 43, 119

absolute value, 18, 26

actual, 119

`all-different`, 18, 29, 42, 43, 119

`automatical posting`, 20

cardinality, 36

conceptual, 119

cumulative, 18

cycle, 18, 48, 52

declaration, 20

definition, 6, 20

disequality, 18

disjunction, 33

disjunctive, 18, 33

distance, 18, 26

distribute, 37, 39

element, 18, 37

equivalence, 18, 42

explicit posting, 21, 67

generic binary, 18, 50

global, 1, 119

global cardinality, 39

implication, 18, 42

implicit, 64

linear, 18

logic, 18, 35, 42

maximum, 14, 18

minimum, 18

name, 20

nonlinear, 18

occurrence, 18, 36–38

or, 33

propagation, 21

redundant, 119

unary, 119

`constraint branching`, 35

`constraint engine`, 120

`constraint network`, 119

`Constraint Programming`, 1

`constraint propagation`, 2, 120

`constraint satisfaction problem`, 119

`constraint solver`, 120

`constraint solving`, 120

`contains`, 63

`continuous variable`, 16

`cost function`, 2

CP, see `Constraint Programming`

`CP dashboard`, 67

`cp_find_next_sol`, 6, 21, 24

`cp_minimize`, 21

`cp_post`, 21, 67

`cp_propagate`, 21, 66

`cp_reset_search`, 28, 58

`cp_schedule`, 64

`cp_set_schedule_search`, 84

`cp_show_prob`, 9

`cp_show_stats`, 10, 63

`cpbranching`, 28

`cpctr`, 20

`cpfloatvar`, 6, 16

`cpresource`, 64

`cptask`, 64

`cpvar`, 6

`cpvarlist`, 61

- create, 12
- critical path, 66
- CSP, see constraint satisfaction problem
- cumulative, 18
- cumulative scheduling, 72
- cycle, 18

D

- data
 - input from file, 10
 - sparse, 12
- data file, 10, 12
- data format
 - sparse, 12
- debugging, 10
- decision variable, 2, 5, 119
 - declaration, 6
 - dynamic array, 12
 - list, 61
 - lower bound, 62
 - name, 10
 - test domain value, 63
 - upper bound, 62
- declarations, 6, 24
- default bound, 12
- DEFAULT_LB, 12
- DEFAULT_UB, 12
- disequality constraint, 6
- disjunction
 - implicit, 68
- disjunctive, 18, 33
- disjunctive scheduling, 68
- distance, 18, 26
- distribute, 18, 37, 39
- div, 50
- domain, 2, 119
 - definition, 6
- domain value
 - next, 62
 - previous, 62
 - test, 63
- domain variable, 2
- DSO search path, 98
- dynamic array, 12
- dynamic set, 12

E

- element, 18, 37, 43
- element constraint
 - 2-dimensional, 43
- empty line, 7
- end, 32
- end-model, 6
- enumeration, 2
 - constraints, 56
 - decision variables, 56
 - task-based, 83
 - variable-based, 83
- enumeration strategy, 2
- equiv, 18, 42
- equivalence constraint, 18, 42
- execute model, 7
- exists, 12
- exit, 6

- extension, 92
- extensions package, 93
- external type, 98

F

- feasible solution, 2
- finite domain constraint problem, 119
- finite domain Constraint programming, 2
- forall, 6
- forming, 7
- forward, 24
- function, 50, 62

G

- generic binary constraint, 50
- generic_binary_constraint, 18, 50
- getlb, 62
- getnext, 62
- getparam, 24, 58
- getprev, 62
- getsol, 7
- getub, 62
- getvar, 62
- global constraint, 1

I

- if, 24
- if-then, 24
- implication constraint, 18, 42
- implicit constraint, 64
- implies, 18, 42
- incrementality, 2
- indentation, 7
- instantiation, 119
 - consistent, 119
- IVE, 2
 - starting, 9
 - user graph, 53

K

- kalis, 1
- Kalis for Mosel, 1
- Kalis library, 93
- KALIS_DISCRETE_RESOURCE, 72
- KALIS_DISJUNCTIONS, 90
- KALIS_FORWARD_CHECKING, 25
- KALIS_GEN_ARC_CONSISTENCY, 25
- KALIS_INITIAL_SOLUTION, 84
- KALIS_INPUT_ORDER, 56
- KALIS_LARGEST_EST, 90
- KALIS_LARGEST_EST, 89
- KALIS_LARGEST_LCT, 90
- KALIS_LARGEST_LST, 89
- KALIS_LARGEST_MAX, 56, 61
- KALIS_LARGEST_MIN, 56
- KALIS_MAX_DEGREE, 28, 56
- KALIS_MAX_TO_MIN, 51, 57, 61
- KALIS_MAXREGRET_LB, 56
- KALIS_MAXREGRET_UB, 56
- KALIS_MIDDLE_VALUE, 57
- KALIS_MIN_TO_MAX, 15, 57
- KALIS_NEAREST_VALUE, 57
- KALIS_OPTIMAL_SOLUTION, 84
- KALIS_RANDOM_VALUE, 57

- KALIS_RANDOM_VARIABLE, 56
- KALIS_SMALLEST_DOMAIN, 15, 56
- KALIS_SMALLEST_ECT, 90
- KALIS_SMALLEST_EST, 89
- KALIS_SMALLEST_LCT, 90
- KALIS_SMALLEST_LST, 89
- KALIS_SMALLEST_MAX, 37, 56
- KALIS_SMALLEST_MIN, 51, 56, 61
- KALIS_TASK_INTERVALS, 90
- KALIS_TIMETABLING, 90
- KALIS_UNARY_RESOURCE, 69
- KALIS_WIDEST_DOMAIN, 56

L

- line break, 7
- list of subroutines, 97, 98
- list of variables, 61
 - entry, 62
- load model, 7
- loop, 6
- lower bound, 62
- LPT heuristic, 105

M

- makespan, 29
- MAX_BACKTRACKS, 57
- MAX_COMPUTATION_TIME, 28, 57, 84
- MAX_DEPTH, 57
- MAX_NODES, 57
- MAX_SOLUTIONS, 58
- maximin problem, 59
- maximization, 13
- maximum, 18, 69
- maximum constraint, 14
- maximum regret, 62
- memory management, 99
- minimization, 13
- minimum, 18
- mod, 50
- model, 119
 - compile, 7
 - data driven, 12
 - execution, 7
 - load, 7
 - run, 7
 - structure, 47
- model, 6
- model parameter, 51
- module
 - listing, 25
 - parameters, 24
- module browser, 25
- Mosel, 1
- Mosel debugger, 10
- Mosel language, 1
- Mosel module, 1
- Mosel Native Interface, 93
- MOSEL_DSO, 98

N

- name
 - constraint, 20
 - variable, 10
- next domain value, 62

NI, see Mosel Native Interface

NODES, 24

O

- objective function, 2
- occurrence, 18, 37, 38
- OPT_ABS_TOLERANCE, 58
- OPT_REL_TOLERANCE, 58
- optimal solution, 2
- optimization, 13, 120
- OPTIMIZE_WITH_RESTART, 58
- or, 33, 35
- output, 6, 9
- overloading, 113

P

- parameters, 51
- posting constraints, 20, 21, 67
- previous domain value, 62
- probe_assign_var, 51, 56
- probe_settle_disjunction, 56
- problem
 - solving, 6
- procedure, 24, 30, 34, 40
- producer-consumer constraint, 79
- propagating constraints, 21
- propagation, 119, 120
- propagation algorithm, 25
- pruning, 25

R

- range set, 6
- resource
 - discrete, 72
 - non-renewable, 75
 - renewable, 75
 - unary, 68
- returned, 62
- run model, 7

S

- scheduling
 - cumulative, 72
 - disjunctive, 68
- search
 - exhaustive, 56
 - incomplete, 56
 - resetting, 28, 58, 60
 - restart, 28, 58, 60
 - time limit, 28
- search methods, 120
- search strategy, 2, 28, 120
 - user defined, 61
- search tree, 16
- set
 - definition, 6
 - range, 6
- set_resource_attributes, 69
- set_task_attributes, 69
- setdomain, 6, 12
- setname
 - tt, 10
- setname, 16
- setparam, 21, 58

- setpredecessors, 67
- setsetuptimes, 81
- setsuccessors, 67
- settarget, 57
- settle_disjunction, 35, 56
- setup time, 81
- solution, 119
 - feasible, 2
 - optimal, 2
 - printing, 6
- solution callback, 28, 58
- solver statistics, 63
- solving, 6
- space, 7
- sparse array, 12
- split_domain, 46, 56
- stopping criteria, 57
- sub-cycle elimination, 42
- subroutine, 24, 30, 34, 40, 47, 50
- sum
 - condition, 11
- symmetry breaking, 28, 75
- system of constraints, 120

T

- target value, 57
- task
 - predecessor, 67
 - successor, 67
- task selection, 89
- task-based
 - enumeration, 83
- task_serialize, 56, 85
- time limit, 28

U

- upper bound, 62
- user constraint, 50
- user search, 61
- uses, 6

V

- value selection strategy, 15, 57
 - user defined, 61
- variable, 119
 - default bounds, 12
- variable domain, 2
- variable selection strategy, 15, 56
 - user defined, 61
- variable-based
 - enumeration, 83
- VERBOSE_LEVEL, 84

W

- while, 24
- write, 7
- writeln, 7

X

- Xpress-IVE, see IVE
- Xpress-Kalis, 1
- Xpress-Mosel, see Mosel
- XPRM_TYP_EXTN, 98