FICO® Xpress Mosel

6.12

USER GUIDE

FICO® Xpress Optimization

FICO

©2001–2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

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

Patent(s): www.fico.com/en/patents

FICO® Xpress Mosel 6.12 (FICO® Xpress 9.9)

Deliverable Version: A

Last Revised: 15 May, 2026

Contents

I Using the Mosel language 1

Introduction 2

Why you need Mosel . . . 2
What you need to know before using Mosel . . . 2
Symbols and conventions . . . 3
The structure of this guide . . . 4

1 Getting started with Mosel 5

1.1 Entering a model . . . 5
1.2 The chess set problem: description . . . 5
1.2.1 A first formulation . . . 5
1.3 Solving the chess set problem . . . 6
1.3.1 Building the model . . . 6
1.3.2 Obtaining a solution using Mosel . . . 7
1.3.3 Running Mosel from a command line . . . 8
1.3.4 Using Xpress Workbench . . . 8

2 Some illustrative examples 11

2.1 The burglar problem . . . 11
2.1.1 Model formulation . . . 11
2.1.2 Implementation . . . 11
2.1.3 The burglar problem revisited . . . 14
2.2 A blending example . . . 15
2.2.1 The model background . . . 15
2.2.2 Model formulation . . . 15
2.2.3 Implementation . . . 16
2.2.4 Re-running the model with new data . . . 17
2.2.5 Reading data from spreadsheets and databases . . . 18
2.2.5.1 Excel spreadsheets . . . 18
2.2.5.2 Database example . . . 20
2.2.5.3 Generic spreadsheet example . . . 21

3 More advanced modeling features 23

3.1 Overview . . . 23
3.2 A transport example . . . 23
3.2.1 Model formulation . . . 23
3.2.2 Implementation . . . 24
3.3 Conditional generation — the | operator . . . 26
3.3.1 Conditional variable creation and `create` . . . 26
3.4 Reading sparse data . . . 28

3.4.1 Data input with `initializations from` . . . 28
3.4.2 Data input with `readln` . . . 28
3.4.3 Data input with `diskdata` . . . 29
3.5 I/O error handling . . . 29

4 Integer Programming 32

4.1 Integer Programming entities in Mosel . . . 32
4.2 A project planning model . . . 35
4.2.1 Model formulation . . . 35
4.2.2 Implementation . . . 36
4.3 The project planning model using Special Ordered Sets . . . 37
4.4 Purchasing with price breaks . . . 38
4.4.1 Model formulation . . . 38
4.4.2 Implementation . . . 39
4.5 The purchasing model using piecewise linear expressions . . . 40

5 Overview of subroutines and reserved words 43

5.1 Modules . . . 43
5.2 Reserved words . . . 45

6 Correcting errors in Mosel models 46

6.1 Correcting syntax errors in Mosel . . . 46
6.2 Correcting run time errors in Mosel . . . 47

II Advanced language features 48

Overview 49

7 Flow control constructs 50

7.1 Selections . . . 50
7.2 Loops . . . 52
7.2.1 `forall` . . . 52
7.2.1.1 Multiple indices . . . 53
7.2.1.2 Conditional looping . . . 53
7.2.1.3 Counters . . . 53
7.2.2 `with` . . . 54
7.2.3 `while` . . . 55
7.2.4 `repeat until` . . . 55

8 Arrays, sets, lists, records, and user types 57

8.1 Arrays . . . 57
8.1.1 Array declaration . . . 58
8.1.1.1 Multiple indices and array access functions . . . 58
8.1.1.2 create . . . 59
8.1.2 Array initialization from file . . . 60
8.1.3 Automatic arrays: the array operator . . . 61
8.2 Initializing sets . . . 61
8.2.1 Constant sets . . . 61
8.2.2 Set initialization from file, finalized and fixed sets . . . 62

8.3 Working with sets . . . 63
8.3.1 Set operators . . . 64
8.4 Initializing lists . . . 65
8.4.1 Constant list . . . 65
8.4.2 List initialization from file . . . 65
8.5 Working with lists . . . 66
8.5.1 Enumeration . . . 66
8.5.2 List operators . . . 67
8.5.3 List handling functions . . . 67
8.6 Records . . . 69
8.6.1 Defining records . . . 69
8.6.2 Initialization of records from file . . . 70
8.7 User types . . . 72
8.7.1 Type constructors . . . 73
8.8 Unions . . . 74
8.8.1 Defining unions . . . 74
8.8.2 Initialization of unions from file . . . 76
8.8.3 Working with the 'reference to' operator . . . 77

9 Functions and procedures 79

9.1 Subroutine definition . . . 79
9.2 Parameters . . . 80
9.2.1 Variable number of arguments . . . 81
9.3 Recursion . . . 82
9.4 `forward` . . . 82
9.5 Overloading of subroutines . . . 83
9.6 Subroutine references . . . 85

10 Output 87

10.1 Producing formatted output . . . 87
10.2 File output . . . 89
10.2.1 Data output with `initializations to` . . . 89
10.2.2 Data output with `writeln` . . . 90
10.2.3 Data output with `diskdata` . . . 90
10.2.4 Solution output with `initializations to` . . . 91
10.3 Real number format . . . 92

11 More about Integer Programming 94

11.1 Cut generation . . . 94
11.1.1 Example problem . . . 94
11.1.2 Model formulation . . . 94
11.1.3 Implementation . . . 95
11.1.4 Cut-and-Branch . . . 97
11.1.5 Comparison tolerance . . . 98
11.1.6 Branch-and-Cut . . . 98
11.2 Column generation . . . 100
11.2.1 Example problem . . . 100
11.2.2 Model formulation . . . 100
11.2.3 Implementation . . . 101
11.2.4 Alternative implementation: Working with multiple problems . . . 104

12 Extensions to Linear Programming 106

12.1 Recursion . . . 106
12.1.1 Example problem . . . 106
12.1.2 Model formulation . . . 106
12.1.3 Implementation . . . 107
12.2 Goal Programming . . . 109
12.2.1 Example problem . . . 109
12.2.2 Implementation . . . 109
12.2.3 Multi-objective optimization with Xpress Optimizer . . . 111

III Working with the Mosel libraries 113

Overview 114

13 C interface 115
13.1 Basic tasks . . . 115
13.1.1 Compiling a model in C . . . 115
13.1.2 Executing a model in C . . . 116
13.1.3 Termination . . . 116
13.2 Parameters . . . 117
13.3 Accessing modeling objects and solution values . . . 117
13.3.1 Accessing sets . . . 117
13.3.2 Retrieving solution values . . . 118
13.3.3 Sparse arrays . . . 119
13.4 Exchanging data between an application and a model . . . 120
13.4.1 Dense arrays . . . 121
13.4.2 Sparse arrays . . . 122
13.4.3 Dynamic data . . . 124
13.4.4 Scalars . . . 127
13.5 Redirecting the Mosel output . . . 129
13.6 Problem solving in C with Xpress Optimizer . . . 130

14 Other programming language interfaces 132
14.1 Java . . . 132
14.1.1 Compiling and executing a model in Java . . . 132
14.1.2 Termination . . . 132
14.1.3 Parameters . . . 133
14.1.4 Accessing sets . . . 133
14.1.5 Retrieving solution values . . . 134
14.1.6 Sparse arrays . . . 135
14.1.7 Exchanging data between an application and a model . . . 136
14.1.7.1 Dense arrays . . . 136
14.1.7.2 Sparse arrays . . . 137
14.1.7.3 Dynamic data . . . 138
14.1.7.4 Scalars . . . 141
14.1.8 Redirecting the Mosel output . . . 142
14.2 .NET . . . 142
14.2.1 Compiling and executing a model in C# . . . 143
14.2.2 Termination . . . 143
14.2.3 Parameters . . . 143
14.2.4 Accessing sets . . . 144
14.2.5 Retrieving solution values . . . 145

14.2.6 Sparse arrays . . . 146
14.2.7 Exchanging data between an application and a model . . . 147
14.2.7.1 Dense arrays . . . 147
14.2.7.2 Sparse arrays . . . 148
14.2.7.3 Dynamic data . . . 150
14.2.7.4 Scalars . . . 152
14.2.8 Redirecting the Mosel output . . . 153
14.3 Python . . . 155
14.3.1 Compiling and executing a model in Python . . . 155
14.3.2 Termination . . . 155
14.3.3 Parameters . . . 155
14.3.4 Accessing sets . . . 156
14.3.5 Retrieving solution values . . . 156
14.3.6 Sparse arrays . . . 157
14.3.7 Exchanging data between an application and a model . . . 157
14.3.7.1 Dense arrays . . . 157
14.3.7.2 Sparse arrays . . . 158
14.3.7.3 Scalars . . . 159
14.3.8 Redirecting the Mosel output . . . 159
14.4 VBA . . . 159
14.4.1 Compiling and executing a model in VBA . . . 159
14.4.2 Parameters . . . 161
14.4.3 Redirecting the Mosel output . . . 162

IV Extensions and tools 164

Overview 165

15 Debugger and Profiler 166

15.1 The Mosel Debugger . . . 166
15.1.1 Using the Mosel Debugger . . . 166
15.1.1.1 Debugging concurrent models . . . 168
15.1.2 Debugger in Xpress Workbench . . . 169
15.2 Efficient modeling through the Mosel Profiler . . . 169
15.2.1 Using the Mosel Profiler . . . 169
15.2.1.1 Profiling concurrent models . . . 171
15.2.2 Other commands for model analysis . . . 171
15.2.3 Some recommendations for efficient modeling . . . 172

16 Packages 174

16.1 Definition of constants . . . 175
16.2 Definition of subroutines . . . 176
16.2.1 Implementation alternative using union types and iterator . . . 177
16.3 Definition of types . . . 178
16.4 Definition of parameters . . . 179
16.5 Namespaces . . . 181
16.6 Package version management . . . 183
16.7 Packages vs. modules . . . 185

17 Language extensions 187

17.1 Generalized file handling . . . 187

17.1.1 Displaying the available I/O drivers . . . 187
17.1.2 List of I/O drivers . . . 188
17.2 Multiple models and parallel solving with *mmjobs* . . . 194
17.2.1 Running a model from another model . . . 195
17.2.2 Compiling to memory . . . 195
17.2.3 Exchanging data between models . . . 196
17.2.4 Distributed computing . . . 197
17.3 Graphics and GUIs . . . 198
17.3.1 Drawing user graphs with *mmsvg* . . . 199
17.3.2 XML and HTML . . . 200
17.3.2.1 mmxml . . . 200
17.3.2.2 Reading and writing XML data . . . 200
17.3.2.3 Generating HTML . . . 203
17.3.3 Xpress Insight . . . 204
17.4 Solvers . . . 207
17.4.1 QCQP solving with Xpress Optimizer . . . 207
17.4.2 Xpress NonLinear and Xpress Global . . . 209
17.4.3 Xpress Kalis . . . 210
17.5 Date and time data types . . . 212
17.5.1 Initializing dates and times . . . 212
17.5.2 Dates and times as constants . . . 214
17.5.3 Conversion to and from numbers . . . 214
17.5.4 Operations and access functions . . . 215
17.6 Text handling and regular expressions . . . 216
17.6.1 `text` *vs.* `string` . . . 217
17.6.2 Parsing text . . . 217
17.6.3 Regular expressions . . . 218

18 Annotations 220

18.1 Accessing annotations . . . 221
18.2 moseldoc . . . 223

V Remote invocation of Mosel 227

Overview 228

19 XPRD C 229

19.1 Exchanging data with the model . . . 230

20 XPRD Java 234

20.1 Exchanging data with the model . . . 235

Appendix 239

A Mosel Language overview 240

A.1 Structure of a Mosel model . . . 240
A.2 Data structures . . . 241
A.3 Selection statements . . . 242
A.4 Loops . . . 243

A.5 Operators . . . 243
A.6 Built in functions and procedures . . . 245
A.7 Constraint handling . . . 246
A.8 Problem handling . . . 247

B Good modeling practice with Mosel 248

B.1 Using constants and parameters . . . 248
B.2 Naming sets . . . 248
B.3 Finalizing sets and dynamic arrays . . . 249
B.4 Ordering indices . . . 251
B.5 Use of `exists` . . . 251
B.6 Structuring a model . . . 252
B.7 Transforming subroutines into user modules . . . 252
B.8 Algorithm choice and parameter settings . . . 252

C Character encoding in Mosel 254

C.1 What is a 'character encoding', 'character map', 'code page'? . . . 254
C.2 What is Unicode? . . . 255
C.3 What is the meaning of UTF-8,16,32 and UCS-2? . . . 255
C.4 What is a BOM? . . . 255
C.5 Which character encoding is configured on my computer? . . . 255
C.6 Which files are concerned by character encoding in Mosel? . . . 256
C.7 How can I convert the character encoding of a text file? . . . 256

D Contacting FICO 258

FICO Customer Support . . . 258
FICO Community . . . 258
Documentation . . . 258
FICO Learning . . . 259
Sales and maintenance . . . 259
About FICO . . . 259

Index 260

I. Using the Mosel language

Introduction

Why you need Mosel

'*Mosel*' is not an acronym. It is pronounced like the German river, mo-zul. It is an advanced modeling and solving language and environment, where optimization problems can be specified and solved with the utmost precision and clarity.

Here are some of the features of Mosel

- Mosel's *easy syntax* is regular and described formally in the reference manual.
- Mosel supports *dynamic objects*, which do not require pre-sizing. For instance, you do not have to specify the maximum sizes of the indices of a variable x.
- Mosel models are *pre-compiled*. Mosel compiles a model into a binary file which can be run on any computer platform, and which hides the intellectual property in the model if so required.
- Mosel is *embeddable*. There is a runtime library which can be called from your favorite programming language if required. You can access any of the model's objects from your programming language.
- Mosel is *easily extended* through the concept of modules. It is possible to write a set of functions, which together stand alone as a module. Several modules are supplied with the Mosel distribution, including Xpress Optimizer.
- Support for *user-written functions* and procedures is provided.
- The use of *sets of objects* is supported.
- Constraints and variables *etc.* can be added *incrementally*. For instance, column generation can depend on the results of previous optimizations, so subproblems are supported.

The modeling component of Mosel provides you with an easy to use yet powerful language for describing your problem. It enables you to gather the problem data from text files and a range of popular spreadsheets and databases, and gives you access to a variety of solvers, which can find optimal or near-optimal solutions to your model.

What you need to know before using Mosel

Before using Mosel you should be comfortable with the use of symbols such as x or y to represent unknown quantities, and the use of this sort of variable in simple linear equations and inequalities, for example:

$$x + y \leqslant 6$$

Experience of a basic course in Mathematical or Linear Programming is worthwhile, but is not essential. Similarly some familiarity with the use of computers would be helpful.

For all but the simplest models you should also be familiar with the idea of summing over a range of variables. For example, if $\mathrm{produce}_j$ is used to represent the number of cars produced on production line j then the total number of cars produced on all N production lines can be written as:

$$\sum_{j=1}^{N} \mathrm{produce}_j$$

This says 'sum the output from each production line $\mathrm{produce}_j$ over all production lines j from $j = 1$ to $j = N$'.

If our target is to produce at least 1000 cars in total then we would write the inequality:

$$\sum_{j=1}^{N} \mathrm{produce}_j \geqslant 1000$$

We often also use a set notation for the sums. Assuming that LINES is the set of production lines $\{1, .., N\}$, we may write equivalently:

$$\sum_{j \in \mathrm{LINES}} \mathrm{produce}_j \geqslant 1000$$

This may be read 'sum the output from each production line $\mathrm{produce}_j$ over all production lines j in the set LINES'.

Other common mathematical symbols that are used in the text are $\mathbb{N}$ (the set of non-negative integer numbers $\{0, 1, 2, ...\}$), $\cap$ and $\cup$ (intersection and union of sets), $\wedge$ and $\vee$ (logical 'and' and 'or'), the all-quantifier $\forall$ (read 'for all'), and $\exists$ (read 'exists').

Mosel closely mimics the mathematical notation an analyst uses to describe a problem. So provided you are happy using the above mathematical notation the step to using a modeling language will be straightforward.

Symbols and conventions

We have used the following conventions within this guide:

- Mathematical objects are presented in *italics*.
- Examples of commands, models and their output are printed in a `Courier font`. Filenames are given in lower case Courier.
- Decision variables have lower case names; in most example problems these are verbs (such as use, take).
- Constraint names start with an upper case letter, followed by mostly lower case (*e.g.* Profit, $\mathrm{TotalCost}$).
- Data (arrays, sets, lists) and constants are written entirely with upper case (*e.g.* DEMAND, COST, ITEMS).
- The vertical bar symbol | is found on many keyboards as a vertical line with a small gap in the middle, but often confusingly displays on-screen without the small gap. In the UNIX world it is referred to as the pipe symbol. (Note that this symbol is not the same as the character sometimes used to draw boxes on a PC screen.) In ASCII, the | symbol is 7C in hexadecimal, 124 in decimal.

The structure of this guide

This user guide is structured into these main parts

- *Part I* describes the use of Mosel for people who want to build and solve Mathematical Programming (MP) problems. These will typically be Linear Programming (LP), Mixed Integer Programming (MIP), or Quadratic Programming (QP) problems. The part has been designed to show the modeling aspects of Mosel, omitting most of the more advanced programming constructs.
- *Part II* is designed to help those users who want to use the powerful programming language facilities of Mosel, using Mosel as a modeling, solving and programming environment. Items covered include looping (with examples), more about using sets, producing nicely formatted output, functions and procedures. We also give some advanced MP examples, including Branch-and-Cut, column generation, Goal Programming and Successive Linear Programming.
- *Part III* shows how Mosel models can be embedded into large applications using programming languages like C, Java, or C#.
- *Part IV* gives examples of some of the advanced features of Mosel, including the use of the Mosel Debugger and Profiler for the development and analysis of large-scale Mosel models, an introduction to the notion of packages, and an overview of the functionality of the modules in the Mosel distribution.

This user guide is deliberately informal and is not complete. It must be read in conjunction with the Mosel reference manual, where features are described precisely and completely.

Chapter 1

Getting started with Mosel

1.1 Entering a model

In this chapter we will take you through a very small manufacturing example to illustrate the basic building blocks of Mosel.

Models are entered into a Mosel file using a standard text editor (do not use a word processor as an editor as this may not produce an ASCII file).

If you have access to Windows, Xpress Workbench is the model development environment to use. The Mosel file is then loaded into Mosel, and compiled. Finally, the compiled file can be run. This chapter will show the stages in action.

1.2 The chess set problem: description

To illustrate the model development and solving process we shall take a very small example.

A joinery makes two different sizes of boxwood chess sets. The smaller size requires 3 hours of machining on a lathe and the larger only requires 2 hours, because it is less intricate. There are four lathes with skilled operators who each work a 40 hour week. The smaller chess set requires 1 kg of boxwood and the larger set requires 3 kg. However boxwood is scarce and only 200 kg per week can be obtained.

When sold, each of the large chess sets yields a profit of $20, and one of the small chess set has a profit of $5. The problem is to decide how many sets of each kind should be made each week to maximize profit.

1.2.1 A first formulation

Within limits, the joinery can *vary* the number of large and small chess sets produced: there are thus two *decision variables* (or simply *variables*) in our model, one decision variable per product. We shall give these variables abbreviated names:

small: the number of small chess sets to make
large: the number of large chess sets to make

The number of large and small chess sets we should produce to achieve the maximum contribution to profit is determined by the optimization process. In other words, we look to the optimizer to tell us the best values of small, and large.

The values which small and large can take will always be *constrained* by some physical or technological limits: they may be constrained to be equal to, less than or greater than some constant. In our case we note that the joinery has a maximum of 160 hours of machine time available per week. Three hours are needed to produce each small chess set and two hours are needed to produce each large set. So the

number of hours of machine time actually used each week is $3 \cdot small + 2 \cdot large$. One constraint is thus:

$$3 \cdot small + 2 \cdot large \leqslant 160 \text{ (lathe-hours)}$$

which restricts the allowable combinations of small and large chess sets to those that do not exceed the lathe-hours available.

In addition, only 200 kg of boxwood is available each week. Since small sets use 1 kg for every set made, against 3 kg needed to make a large set, a second constraint is:

$$1 \cdot small + 3 \cdot large \leqslant 200 \text{ (kg of boxwood)}$$

where the left hand side of the inequality is the amount of boxwood we are planning to use and the right hand side is the amount available.

The joinery cannot produce a negative number of chess sets, so two further *non-negativity constraints* are:

$$small \geqslant 0$$
$$large \geqslant 0$$

In a similar way, we can write down an expression for the total profit. Recall that for each of the large chess sets we make and sell we get a profit of $20, and one of the small chess set gives us a profit of $5. The total profit is the sum of the individual profits from making and selling the $small$ small sets and the $large$ large sets, *i.e.*

$$\text{Profit} = 5 \cdot small + 20 \cdot large$$

Profit is the *objective function*, a linear function which is to be optimized, that is, maximized. In this case it involves all of the decision variables but sometimes it involves just a subset of the decision variables. In maximization problems the objective function usually represents profit, turnover, output, sales, market share, employment levels or other 'good things'. In minimization problems the objective function describes things like total costs, disruption to services due to breakdowns, or other less desirable process outcomes.

The collection of variables, constraints and objective function that we have defined are our *model*. It has the form of a *Linear Programming problem*: all constraints are linear equations or inequalities, the objective function also is a linear expression, and the variables may take any non-negative real value.

1.3 Solving the chess set problem

1.3.1 Building the model

The Chess Set problem can be solved easily using Mosel. The first stage is to get the model we have just developed into the syntax of the Mosel language. Remember that we use the notation that items in italics (for example, $small$) are the mathematical variables. The corresponding Mosel variables will be the same name in non-italic courier (for example, `small`).

We illustrate this simple example by using the command line version of Mosel. The model can be entered into a file named, perhaps, `chess.mos` as follows:

```
model "Chess"
 declarations
  small: mpvar                      ! Number of small chess sets to make
  large: mpvar                      ! Number of large chess sets to make
 end-declarations

 Profit:=  5*small + 20*large        ! Objective function
```

```
 Lathe:=   3*small + 2*large <= 160  ! Lathe-hours
 Boxwood:=   small + 3*large <= 200  ! kg of boxwood
end-model
```

Indentations are purely for clarity. The symbol `!` signifies the start of a *comment*, which continues to the end of the line. Comments over multiple lines start with `(!` and terminate with `!)`.

Notice that the character '`*`' is used to denote multiplication of the decision variables by the units of machine time and wood that one unit of each uses in the `Lathe` and `Boxwood` constraints.

The modeling language distinguishes between upper and lower case, so `Small` would be recognized as different from `small`.

Let's see what this all means.

A *model* is enclosed in a `model` / `end-model` block.

The *decision variables* are declared as such in the `declarations` / `end-declarations` block. Every decision variable must be declared. LP, MIP and QP variables are of type `mpvar`. Several decision variables can be declared on the same line, so

```
 declarations
  small, large: mpvar
 end-declarations
```

is exactly equivalent to what we first did. By default, Mosel assumes that all `mpvar` variables are constrained to be non-negative unless it is informed otherwise, so there is no need to specify non-negativity constraints on variables.

Here is an example of a *constraint*:

```
Lathe:=   3*small + 2*large <= 160
```

The name of the constraint is `Lathe`. The actual constraint then follows. If the 'constraint' is unconstrained (for example, it might be an *objective function*), then there is no <=, >= or = part.

In Mosel you enter the entire model before starting to compile and run it. Any errors will be signaled when you try to compile the model, or later when you run it (see Chapter 6 on correcting syntax errors).

1.3.2 Obtaining a solution using Mosel

So far, we have just specified a model to Mosel. Next we shall try to solve it. The first thing to do is to specify to Mosel that it is to use Xpress Optimizer to solve the problem. Then, assuming we can solve the problem, we want to print out the optimum values of the decision variables, `small` and `large`, and the value of the objective function. The model becomes

```
model "Chess (completed)"
 uses "mmxprs"                       ! We shall use Xpress Optimizer</p>

 declarations
  small,large: mpvar                 ! Decision variables: produced quantities
 end-declarations

 Profit:=  5*small + 20*large        ! Objective function
 Lathe:=   3*small + 2*large <= 160  ! Lathe-hours
 Boxwood:=   small + 3*large <= 200  ! kg of boxwood

 maximize(Profit)                    ! Solve the problem

 writeln("Make ", getsol(small), " small sets")
 writeln("Make ", getsol(large), " large sets")
 writeln("Best profit is ", getobjval)
end-model
```

The line

```
uses "mmxprs"
```

tells Mosel that Xpress Optimizer will be used to solve the LP. The Mosel modules `mmxprs` module provides us with such things as maximization, handling bases *etc.*

The line

```
maximize(Profit)
```

tells Mosel to maximize the objective function called `Profit`.

More complicated are the `writeln` statements, though it is actually quite easy to see what they do. If some text is in quotation marks, then it is written literally. `getsol` and `getobjval` are special Mosel functions that return respectively the optimal value of the argument, and the optimal objective function value. `writeln` writes a line terminator after writing all its arguments (to continue writing on the same line, use `write` instead). `writeln` can take many arguments. The statement

```
writeln("small: ", getsol(small), " large: ", getsol(large) )
```

will result in the values being printed all on one line.

1.3.3 Running Mosel from a command line

When you have entered the complete model into a file (let us call it `chess.mos`), we can proceed to get the solution to our problem. We start Mosel at the command prompt by typing the following command

```
mosel execute chess.mos
```

and we will see output something like that below.

```
Make 0 small sets
Make 66.6667 large sets
Best profit is 1333.33
```

The Mosel command for executing the model can be abbreviated to

```
mosel exec chess
```

or simply

```
mosel chess
```

The model execution performed by the command `execute` comprises three stages:

1. Compiling `chess.mos`
2. Loading the compiled model
3. Running the model we have just loaded.

Instead of using `execute`, we can choose to explicitly generate the compiled model file `chess.bim`

```
mosel compile chess.mos
```

followed by

```
mosel run chess.bim
```

to load and run the compiled model.

1.3.4 Using Xpress Workbench

Under Microsoft Windows you may also use Xpress Workbench, a development studio type environment for working with your Mosel models. Xpress Workbench is a complete modeling and optimization development environment that presents Mosel in an easy-to-use Graphical User Interface (GUI), with a built-in text editor.

To execute the model file `chess.mos` you need to carry out the following steps.

- Start up Workbench.
- Open the model file by choosing *File* ≫ *Open*. The model source is then displayed in the central window (the *Workbench Editor*).
- Click the *Run* button at the top of the window, making sure that the desired filename is selected in the input field to its left, or alternatively, choose *Run* ≫ *Run chess.mos*.

The resulting screen display is shown in Figure 1.1.

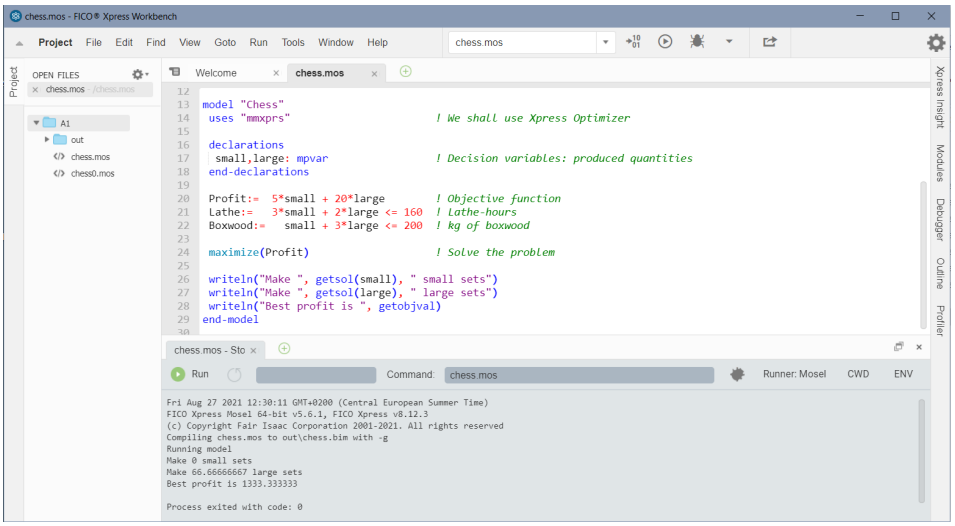


Figure 1.1: Xpress Workbench screen after running chess.mos

The logging pane at the bottom of the workspace is automatically displayed when compilation starts. 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. If the model has been compiled successfully, this pane displays the output produced by running the model.

If the model is run in debug mode by selecting the *Debug* button Workbench makes all information about the solution available through the *Debugger* pane on the right border of the workspace window (see Figure 1.2). By expanding the *Variables* entry in this pane, the solution and reduced cost values for decision variables are displayed. Dual and slack values for constraints may also be obtained.

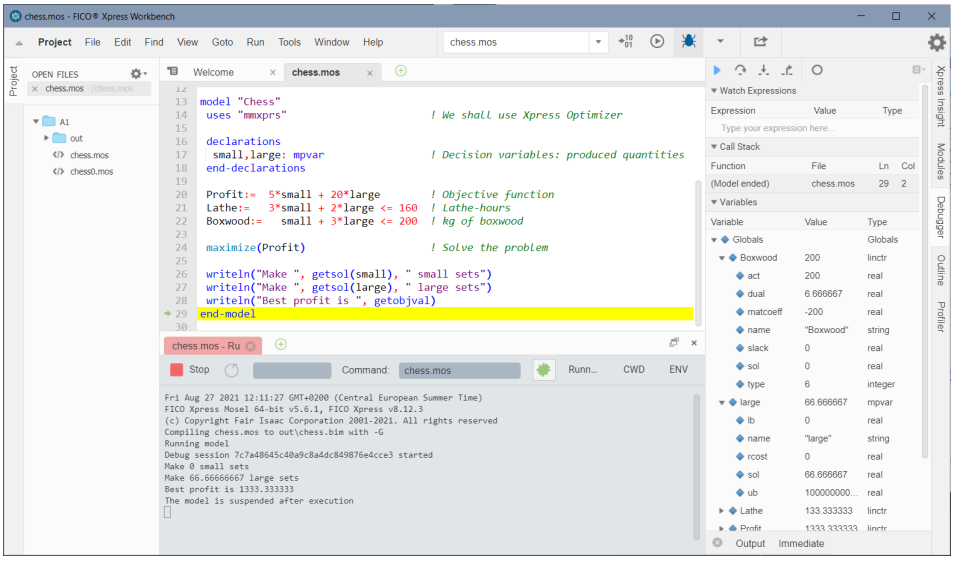


Figure 1.2: Running chess.mos with Xpress Workbench in debug mode

CHAPTER 2

Some illustrative examples

This chapter develops the basics of modeling set out in Chapter 1. It presents some further examples of the use of Mosel and introduces new features:

- **Use of subscripts:** Almost all models of any size have subscripted variables. We show how to define arrays of data and decision variables, introduce the different types of sets that may be used as index sets for these arrays, and also simple loops over these sets.
- **Working with data files:** Mosel provides facilities to read from and write to data files in text format and also from other data sources (databases and spreadsheets).

2.1 The burglar problem

A burglar sees 8 items, of different worths and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum $WTMAX$ he can carry.

2.1.1 Model formulation

We introduce binary variables take_i for all i in the set of all items (ITEMS) to represent the decision whether item i is taken or not. take_i has the value 1 if item i is taken and 0 otherwise. Furthermore, let VALUE_i be the value of item i and WEIGHT_i its weight. A mathematical formulation of the problem is then given by:

$$\text{maximize} \sum_{i \in \mathrm{ITEMS}} \mathrm{VALUE}_i \cdot \mathrm{take}_i$$

$$\sum_{i \in \mathrm{ITEMS}} \mathrm{WEIGHT}_i \cdot \mathrm{take}_i \leqslant WTMAX \quad \text{(weight restriction)}$$

$$\forall i \in \mathrm{ITEMS} : \mathrm{take}_i \in \{0, 1\}$$

The objective function is to maximize the total value, that is, the sum of the values of all items taken. The only constraint in this problem is the weight restriction. This problem is an example of a *knapsack problem*.

2.1.2 Implementation

It may be implemented with Mosel as follows (model file `burglar.mos`):

```
model Burglar
 uses "mmxprs"

 declarations
  WTMAX = 102                    ! Maximum weight allowed
```

```
  ITEMS = 1..8                    ! Index range for items

  VALUE: array(ITEMS) of real     ! Value of items
  WEIGHT: array(ITEMS) of real    ! Weight of items

  take: array(ITEMS) of mpvar     ! 1 if we take item i; 0 otherwise
 end-declarations

! Item:      1   2   3   4   5   6   7   8
  VALUE :: [15, 100, 90, 60, 40, 15, 10,  1]
  WEIGHT:: [ 2,  20, 20, 30, 40, 30, 60, 10]

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                  ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

When running this model we get the following output:

```
Solution:
 Objective: 280
 take(1): 1
 take(2): 1
 take(3): 1
 take(4): 1
 take(5): 0
 take(6): 1
 take(7): 0
 take(8): 0
```

In this model there are a lot of new features, which we shall now explain.

- *Constants:*

  ```
  WTMAX=102
  ```

  declares a constant called `WTMAX`, and gives it the value 102. Since 102 is an integer, `WTMAX` is an integer constant. Anything that is given a value in a declarations block is a constant.

- *Ranges:*

  ```
  ITEMS = 1..8
  ```

  defines a *range set*, that is, a set of consecutive integers from 1 to 8. This range is used as an *index set* for the data arrays (`VALUE` and `WEIGHT`) and for the array of decision variables `take`.

- *Arrays:*

  ```
  VALUE: array(ITEMS) of real
  ```

  defines a one-dimensional array of real values indexed by the range `ITEMS`. Exactly equivalent would be

  ```
  VALUE: array(1..8) of real    ! Value of items
  ```

  Multi-dimensional arrays are declared in the obvious way *e.g.*

```
  VAL3: array(ITEMS, 1..20, ITEMS) of real
```

declares a 3-dimensional real array. Arrays of decision variables (type `mpvar`) are declared likewise, as shown in our example:

```
  x: array(ITEMS) of mpvar
```

declares an array of decision variables `take(1)`, `take(2)`, ..., `take(8)`.

All objects (scalars and arrays) declared in Mosel are always initialized with a default value:

`real`, `integer`: 0
`boolean`: `false`
`string`: `''` (*i.e.* the empty string)

In Mosel, reals are double precision.

- *Assigning values to arrays:*

  The values of data arrays may either be defined in the model as we show in the example or initialized from file (see Section 2.2).

```
  VALUE :: [15, 100, 90, 60, 40, 15, 10,  1]
```

fills the `VALUE` array as follows:
`VALUE(1)` gets the value 15; `VALUE(2)` gets the value 100; ..., `VALUE(8)` gets the value 1.

For a 2-dimensional array such as

```
declarations
 EE: array(1..2, 1..3) of real
end-declarations
```

we might write

```
EE:: [11, 12, 13,
      21, 22, 23 ]
```

which of course is the same as

```
EE:: [11, 12, 13, 21, 22, 23]
```

but much more intuitive. Mosel places the values in the tuple into `EE` 'going across the rows', with the last subscript varying most rapidly. For higher dimensions, the principle is the same. If the index sets of an array are other than ranges they must be given when initializing the array with data, in the case of ranges this is optional. Equivalently to the above we may write

```
VALUE :: (ITEMS)[15, 100, 90, 60, 40, 15, 10,  1]
EE:: (1..2, 1..3)[11, 12, 13,21, 22, 23 ]
```

or even initialize the two-dimensional array `EE` rowwise:

```
EE:: (1, 1..3)[11, 12, 13]
EE:: (2, 1..3)[21, 22, 23 ]
```

- *Summations:*

```
  MaxVal:= sum(i in Items) VALUE(i)*x(i)
```

defines a linear expression called `MaxVal` as the sum

$$\sum_{i \in Items} VALUE_i \cdot x_i$$

- *Naming constraints:*

  Optionally, constraints may be named (as in the chess set example). In the remainder of this manual, we shall name constraints only if we need to refer to them at other places in the model. In most examples, only the objective function is named (here `MaxVal`) — to be able to refer to it in the call to the optimization (here `maximize(MaxVal)`).

- *Simple looping:*

```
  forall(i in ITEMS) take(i) is_binary
```

illustrates looping over all values in an index range. Recall that the index range `ITEMS` is 1, ..., 8, so the statement says that `take(1)`, `take(2)`, ..., `take(8)` are all binary variables.
There is another example of the use of `forall` at the penultimate line of the model when writing out all the solution values.

- *Integer Programming variable types:*

  To make an `mpvar` variable, say variable `xbinvar`, into a binary (0/1) variable, we just have to say

```
  xbinvar is_binary
```

  To make an `mpvar` variable an integer variable, *i.e.* one that can only take on integral values in a MIP problem, we would have

```
  xintvar is_integer
```

2.1.3 The burglar problem revisited

Consider this model (`burglari.mos`):

```
model "Burglar (index set)"
 uses "mmxprs"

 declarations
  WTMAX = 102                    ! Maximum weight allowed
  ITEMS = {"camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"}     ! Index set for items

  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items

  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

 VALUE("camera")  := 15;  WEIGHT("camera")  :=  2
 VALUE("necklace"):=100;  WEIGHT("necklace"):= 20
 VALUE("vase")    := 90;  WEIGHT("vase")    := 20
 VALUE("picture") := 60;  WEIGHT("picture") := 30
 VALUE("tv")      := 40;  WEIGHT("tv")      := 40
 VALUE("video")   := 15;  WEIGHT("video")   := 30
 VALUE("chest")   := 10;  WEIGHT("chest")   := 60
 VALUE("brick")   :=  1;  WEIGHT("brick")   := 10

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

What have we changed? The answer is, 'not very much'.

- *String indices:*

```
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}
```

declares that this time `ITEMS` is a *set of strings*. The indices now take the string values 'camera', 'necklace' *etc.* Since string index sets have no fixed ordering like the range set we have used in the first version of the model, we now need to initialize every data item separately, or alternatively, write out the index sets when defining the array values, such as

```
VALUE :: (["camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"])[15,100,90,60,40,15,10,1]
WEIGHT:: (["camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"])[2,20,20,30,40,30,60,10]
```

If we run the model, we get

```
Solution:
 Objective: 280
 take(camera): 1
 take(necklace): 1
 take(vase): 1
 take(picture): 1
 take(tv): 0
 take(video): 1
 take(chest): 0
 take(brick): 0
```

- *Continuation lines:*

  Notice that the statement

```
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}
```

  was spread over two lines. Mosel is smart enough to recognize that the statement is not complete, so it automatically tries to continue on the next line. If you wish to extend a single statement to another line, just cut it after a symbol that implies a continuation, like an operator (+, –, <=, ...) or a comma (`,`) in order to warn the analyzer that the expression continues in the following line(s). For example

```
ObjMax:= sum(i in Irange, j in Jrange) TAB(i,j) * x(i,j) +
          sum(i in Irange) TIB(i) * delta(i)             +
          sum(j in Jrange) TUB(j) * phi(j)
```

  Conversely, it is possible to place several statements on a single line, separating them by semicolons (like `x1 <= 4; x2 >= 7`).

2.2 A blending example

2.2.1 The model background

A mining company has two types of ore available: Ore 1 and Ore 2. The ores can be mixed in varying proportions to produce a final product of varying quality. For the product we are interested in, the 'grade' (a measure of quality) of the final product must lie between the specified limits of 4 and 5. It sells for $\mathrm{REV} =$ £125 per ton. The costs of the two ores vary, as do their availabilities. The objective is to maximize the total net profit.

2.2.2 Model formulation

Denote the amounts of the ores to be used by use_1 and use_2. Maximizing net profit (*i.e.*, sales revenue less cost COST_o of raw material) gives us the objective function:

$$\sum_{o \in \mathrm{ORES}} (\mathrm{REV} - \mathrm{COST}_o) \cdot \mathrm{use}_o$$

We then have to ensure that the grade of the final ore is within certain limits. Assuming the grades of the ores combine linearly, the grade of the final product is:

$$\frac{\sum_{o \in ORES} GRADE_o \cdot use_o}{\sum_{o \in ORES} use_o}$$

This must be greater than or equal to 4 so, cross-multiplying and collecting terms, we have the constraint:

$$\sum_{o \in ORES} (GRADE_o - 4) \cdot use_o \geqslant 0$$

Similarly the grade must not exceed 5.

$$\frac{\sum_{o \in ORES} GRADE_o \cdot use_o}{\sum_{o \in ORES} use_o} \leqslant 5$$

So we have the further constraint:

$$\sum_{o \in ORES} (5 - GRADE_o) \cdot use_o \geqslant 0$$

Finally only non-negative quantities of ores can be used and there is a limit to the availability $AVAIL_o$ of each of the ores. We model this with the constraints:

$$\forall o \in ORES : 0 \leqslant use_o \leqslant AVAIL_o$$

2.2.3 Implementation

The above problem description sets out the relationships which exist between variables but contains few explicit numbers. Focusing on relationships rather than figures makes the model much more flexible. In this example only the selling price REV and the upper/lower limits on the grade of the final product ($MINGRADE$ and $MAXGRADE$) are fixed.

Enter the following model into a file `blend.mos`.

```
model "Blend"
 uses "mmxprs"

 declarations
  REV = 125                      ! Unit revenue of product
  MINGRADE = 4                   ! Minimum permitted grade of product
  MAXGRADE = 5                   ! Maximum permitted grade of product
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar      ! Quantities of ores used
 end-declarations

! Read data from file blend.dat
 initializations from 'blend.dat'
  COST
  AVAIL
  GRADE
 end-initializations

! Objective: maximize total profit
 Profit:= sum(o in ORES) (REV-COST(o))* use(o)
```

```
! Lower and upper bounds on ore quality
 sum(o in ORES) (GRADE(o)-MINGRADE)*use(o) >= 0
 sum(o in ORES) (MAXGRADE-GRADE(o))*use(o) >= 0

! Set upper bounds on variables (lower bound 0 is implicit)
 forall(o in ORES) use(o) <= AVAIL(o)

 maximize(Profit)                ! Solve the LP-problem

 ! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(o in ORES)  writeln(" use(" + o + "): ", getsol(use(o)))

end-model
```

The file `blend.dat` contains the following:

```
! Data file for 'blend.mos'
COST: [85 93]
AVAIL: [60 45]
GRADE: [2.1 6.3]
```

The `initializations from` / `end-initializations` block is new here, telling Mosel where to get data from to initialize named arrays. The order of the data items in the file does not have to be the same as that in the initializations block; equally acceptable would have been the statements

```
initializations from 'blend.dat'
 AVAIL GRADE COST
end-initializations
```

Alternatively, since all data arrays have the same indices, they may be given in the form of a single record, such as `BLENDDATA` in the following data file `blendb.dat`:

```
        !   [COST AVAIL GRADE]
BLENDDATA: [ [85   60   2.1]
             [93   45   6.3] ]
```

In the `initializations` block we need to indicate the label of the data record and in which order the data of the three arrays is given:

```
initializations from 'blendb.dat'
 [COST,AVAIL,GRADE] as 'BLENDDATA'
end-initializations
```

2.2.4 Re-running the model with new data

There is a problem with the model we have just presented — the name of the file containing the costs date is hard-wired into the model. If we wanted to use a different file, say `blend2.dat`, then we would have to edit the model, and recompile it.

Mosel has *parameters* to help with this situation. A model parameter is a symbol, the value of which can be set just before running the model, often as an argument of the `run` command of the command line interpreter.

```
model "Blend 2"
 uses "mmxprs"

 parameters
  DATAFILE="blend.dat"
 end-parameters

 declarations
```

```
  REV = 125                       ! Unit revenue of product
  MINGRADE = 4                    ! Minimum permitted grade of product
  MAXGRADE = 5                    ! Maximum permitted grade of product
  ORES = 1..2                     ! Range of ores

  COST: array(ORES) of real       ! Unit cost of ores
  AVAIL: array(ORES) of real      ! Availability of ores
  GRADE: array(ORES) of real      ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar       ! Quantities of ores used
 end-declarations

! Read data from file
 initializations from DATAFILE
  COST
  AVAIL
  GRADE
 end-initializations

 ...

end-model
```

The parameter `DATAFILE` is recognized as a string, and its default value is specified. If we have previously compiled the model into say `blend2.bim`, then the command

```
mosel run blend2 DATAFILE="blend2.dat"
```

will read the cost data from the file we want. Or to compile, load, and run the model using a single command:

```
mosel exec blend2 DATAFILE="blend2.dat"
```

Notice that a model only takes a single `parameters` block that must follow immediately after the `uses` statement(s) at the beginning of the model.

2.2.5 Reading data from spreadsheets and databases

It is quite easy to create and maintain data tables in text files but in many industrial applications data are provided in the form of spreadsheets or need to be extracted from databases. So there is a facility in Mosel whereby the contents of ranges within spreadsheets may be read into data tables and databases may be accessed.

In addition to the documentation of the Mosel modules `mmodbc` and `mmsheet` in the *Mosel language reference manual*, you will find further detail and examples of using the SQL/ODBC and spreadsheet interfaces in other documents of the Xpress distribution: the whitepaper *Using ODBC and other database interfaces with Mosel* explains how to set up an ODBC connection and discusses a large number of examples showing different SQL/ODBC features; the whitepaper *Generalized file handling in Mosel* also contains several examples of the use of ODBC. To give you a flavor of how Mosel's ODBC and spreadsheet interfaces may be used, we now read the data of the blending problem from a spreadsheet and then later from a database.

The ODBC technology is a generic means for accessing databases and some spreadsheets such as certain versions of Microsoft Excel also support (a reduced set of) ODBC functionality. Mosel also provides a specific interface to Excel spreadsheets, an example of which is shown below (Section 2.2.5.1). This interface that supports all basic tasks of data exchange should be used for working with Excel data. A generic alternative for working with spreadsheets in .xls, .xlsx, or .csv format, including on non-Windows platforms, is discussed in Section 2.2.5.3.

2.2.5.1 Excel spreadsheets

Let us suppose that in a Microsoft Excel spreadsheet called `blend.xls` you have inserted the following

into the cells indicated:

Table 2.1: Spreadsheet example data

	A	B	C	D	E	F
1						
2		ORES	COST	AVAIL	GRADE	
3		1	85	60	2.1	
4		2	93	45	6.3	
5						

and called the range B3:E4 `MyRange`.

The following model reads the data for the arrays `COST`, `AVAIL`, and `GRADE` from the Excel range `MyRange`. Note that we have added `"mmsheet"` to the `uses` statement to indicate that we are using the Mosel spreadsheet module.

```
model "Blend 3"
 uses  "mmxprs", "mmsheet"

 declarations
  REV = 125                      ! Unit revenue of product
  MINGRADE = 4                   ! Minimum permitted grade of product
  MAXGRADE = 5                   ! Maximum permitted grade of product
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar      ! Quantities of ores used
 end-declarations

! Read data from spreadsheet blend.xls
 initializations from "mmsheet.excel:blend.xls"
  [COST,AVAIL,GRADE] as "MyRange"
 end-initializations

 ...

end-model
```

Instead of naming the ranges in the spreadsheet it is equally possible to work directly with the cell references (including the worksheet name, that is, '`Sheet1`' in our case):

```
 initializations from "mmsheet.excel:blend.xls"
  [COST,AVAIL,GRADE] as "[Sheet1$B3:E4]"
 end-initializations
```

or alternatively, work with row and column counters:

```
 initializations from "mmsheet.excel:blend.xls"
  [COST,AVAIL,GRADE] as "[Sheet1$R3C2:R4C5]"
 end-initializations
```

And we can also select specific columns from a range:

```
 initializations from "mmsheet.excel:blend.xls"
  GRADE as "MyRange(#1,#4)"
 end-initializations
```

If the range definition contains the header line with column titles (so, `MyRangeWithHeader` is the area B2:E4) we can also select specific columns via their titles:

```
 initializations from "mmsheet.excel:blend.xls"
  GRADE as "skiph;MyRangeWithHeader(ORES,GRADE)"
 end-initializations
```

2.2.5.2 Database example

If we use Microsoft Access, we might have set up an ODBC *DSN* called `MSAccess`. **Note:** this is where to check whether the DSN is set up with Windows 2000 or XP: *Start* ≫ *Settings* ≫ *Control Panel* ≫ *Administrative Tools* ≫ *Data Sources (ODBC)* ≫ *ODBC drivers*; on Windows 10 open the *ODBC Data Source Administrator*.
Suppose we are extracting data from a table called `MyTable` in the database `blend.mdb`. There are just the four columns `ORES`, `COST`, `AVAIL`, and `GRADE` in `MyTable`, and the data are the same as in the Excel example above. We modify the example above to be

Note: as an alternative to working with Microsoft Access, you can use the built-in SQLite database that is available on all platforms supported by Mosel without any additional installation: replace the filename `blend.mdb` by `blend.sqlite` in the following examples.

```
model "Blend 4"
 uses  "mmxprs", "mmodbc"

 declarations
  REV = 125                     ! Unit revenue of product
  MINGRADE = 4                  ! Minimum permitted grade of product
  MAXGRADE = 5                  ! Maximum permitted grade of product
  ORES = 1..2                   ! Range of ores

  COST: array(ORES) of real     ! Unit cost of ores
  AVAIL: array(ORES) of real    ! Availability of ores
  GRADE: array(ORES) of real    ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar     ! Quantities of ores used
 end-declarations

! Read data from database blend.mdb
 initializations from "mmodbc.odbc:blend.mdb"
  [COST,AVAIL,GRADE] as "MyTable"
 end-initializations

 ...

end-model
```

With ODBC, we can use the field names to select specific columns from a table:

```
 initializations from "mmodbc.odbc:blend.mdb"
  GRADE as "MyTable(ORES,GRADE)"
 end-initializations
```

Instead of using the `initializations` block that automatically generates SQL commands for reading and writing data it is also possible to employ SQL statements in Mosel models. The `initializations` block in the model above is equivalent to the following sequence of SQL statements:

```
 SQLconnect('DSN=MSAccess; DBQ=blend.mdb')
 if not getparam("SQLsuccess"): exit(1)
 SQLexecute("select * from MyTable ", [COST,AVAIL,GRADE])
 SQLdisconnect
```

The SQL statement `"select * from MyTable"` says 'select everything from the table called `MyTable`'. By using SQL statements directly in the Mosel model it is possible to have much more complex selection statements than the ones we have used. Note that it is recommended to check the

value of `SQLsuccess` after calls to individual SQL commands.

When working with a SQLite database it is sufficient to specify the database filename in the `SQLconnect` statement:

```
 SQLconnect('blend.sqlite')
```

To use other databases, for instance a *mysql* database (let us call it `blend`), we merely need to modify the connection string — provided that we have given the same names to the data table and its columns:

```
 initializations from "mmodbc.odbc:DSN=mysql;DB=blend"
```

ODBC, just like Mosel's text file format, may also be used to output data. The reader is referred to the ODBC/SQL documentation and the Xpress whitepaper *Using ODBC and other database interfaces with Mosel* for more detail.

2.2.5.3 Generic spreadsheet example

We shall work once more with the Microsoft Excel spreadsheet called `blend.xls` shown in Table 2.1 where we have defined the range B3:E4 `MyRange`.

This spreadsheet can be accessed via MS Excel as shown above. However, this access method is only available on platforms where Excel is installed. The module *mmsheet* also provides more generic interfaces for working with .xsl, .xlsx and CSV format files (usable, for example, under Linux or MacOS). The corresponding Mosel model looks as follows.

```
model "Blend 3 (spreadsheet)"
 uses "mmsheet", "mmxprs"

 declarations
  REV = 125                     ! Unit revenue of product
  MINGRADE = 4                  ! Minimum permitted grade of product
  MAXGRADE = 5                  ! Maximum permitted grade of product
  ORES = 1..2                   ! Range of ores

  COST: array(ORES) of real     ! Unit cost of ores
  AVAIL: array(ORES) of real    ! Availability of ores
  GRADE: array(ORES) of real    ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar     ! Quantities of ores used
 end-declarations

! Read data from spreadsheet blend.xls
 initializations from "mmsheet.xls:blend.xls"
  [COST,AVAIL,GRADE] as "MyRange"
 end-initializations

 ...

end-model
```

The only modification we have made is quite subtle: in the filename we have replaced `mmsheet.excel` by `mmsheet.xls`.

Variant: Assuming that we have saved the data from our spreadsheet into the CSV format file `blend.csv`, we need to switch to the CSV interface for accessing this data file. A CSV file contains a single worksheet and it is not possible to define named ranges. We therefore now refer directly to the cells via the cell references (similarly to what has been shown for Excel in Section 2.2.5.1 but without stating a sheet name):

```
 initializations from "mmsheet.csv:blend.csv"
  [COST,AVAIL,GRADE] as "[B3:E4]"
 end-initializations
```

or alternatively, using row and column counters:

```
initializations from "mmsheet.csv:blend.csv"
 [COST,AVAIL,GRADE] as "[R3C2:R4C5]"
end-initializations
```

CHAPTER 3

More advanced modeling features

3.1 Overview

This chapter introduces some more advanced features of the modeling language in Mosel. We shall not attempt to cover all its features or give the detailed specification of their formats. These are covered in greater depth in the Mosel Reference Manual.

Almost all large scale LP and MIP problems have a property known as *sparsity*, that is, each variable appears with a non-zero coefficient in a very small fraction of the total set of constraints. Often this property is reflected in the data tables used in the model in that many values of the tables are zero. When this happens, it is more convenient to provide just the non-zero values of the data table rather than listing all the values, the majority of which are zero. This is also the easiest way to input data into data tables with more than two dimensions. An added advantage is that less memory is used by Mosel.

The main areas covered in this chapter are related to this property:

- dynamic arrays
- sparse data
- conditional generation
- displaying data

We start again with an example problem. The following sections deal with the different topics in more detail.

3.2 A transport example

A company produces the same product at different plants in the UK. Every plant has a different production cost per unit and a limited total capacity. The customers (grouped into customer regions) may receive the product from different production locations. The transport cost is proportional to the distance between plants and customers, and the capacity on every delivery route is limited. The objective is to minimize the total cost, whilst satisfying the demands of all customers.

3.2.1 Model formulation

Let PLANT be the set of plants and REGION the set of customer regions. We define decision variables $\mathrm{flow}_{\mathrm{pr}}$ for the quantity transported from plant p to customer region r. The total cost of the amount of product p delivered to region r is given as the sum of the transport cost (the distance between p and r multiplied by a factor $\mathrm{FUELCOST}$) and the production cost at plant p:

$$\text{minimize} \sum_{\mathrm{p}\in\mathrm{PLANT}} \sum_{\mathrm{r}\in\mathrm{REGION}} (\mathrm{FUELCOST}\cdot\mathrm{DISTANCE}_{\mathrm{pr}} + \mathrm{PLANTCOST}_{\mathrm{p}})\cdot \mathrm{flow}_{\mathrm{pr}}$$

The limits on plant capacity are give through the constraints

$$\forall p \in \text{PLANT} : \sum_{r \in \text{REGION}} flow_{pr} \leqslant \text{PLANTCAP}_p$$

We want to meet all customer demands:

$$\forall r \in \text{REGION} : \sum_{p \in \text{PLANT}} flow_{pr} = \text{DEMAND}_r$$

The transport capacities on all routes are limited and there are no negative flows:

$$\forall p \in \text{PLANT}, r \in \text{REGION} : 0 \leqslant flow_{pr} \leqslant \text{TRANSCAP}_{pr}$$

For simplicity's sake, in this mathematical model we assume that all routes $p \to r$ are defined and that we have $\text{TRANSCAP}_{pr} = 0$ to indicate that a route cannot be used.

3.2.2 Implementation

This problem may be implemented with Mosel as shown in the following (model file `transport.mos`):

```
model Transport
 uses "mmxprs"

 declarations
  REGION: set of string                ! Set of customer regions
  PLANT: set of string                 ! Set of plants

  DEMAND: array(REGION) of real        ! Demand at regions
  PLANTCAP: array(PLANT) of real       ! Production capacity at plants
  PLANTCOST: array(PLANT) of real      ! Unit production cost at plants
  TRANSCAP: dynamic array(PLANT,REGION) of real
                                       ! Capacity on each route plant->region
  DISTANCE: dynamic array(PLANT,REGION) of real
                                       ! Distance of each route plant->region
  FUELCOST: real                       ! Fuel cost per unit distance

  flow: dynamic array(PLANT,REGION) of mpvar    ! Flow on each route
 end-declarations

 initializations from 'transprt.dat'
  DEMAND
  [PLANTCAP,PLANTCOST] as 'PLANTDATA'
  [DISTANCE,TRANSCAP] as 'ROUTES'
  FUELCOST
 end-initializations

! Create the flow variables that exist
 forall(p in PLANT, r in REGION | exists(TRANSCAP(p,r)) ) create(flow(p,r))

! Objective: minimize total cost
 MinCost:= sum(p in PLANT, r in REGION | exists(flow(p,r)))
            (FUELCOST * DISTANCE(p,r) + PLANTCOST(p)) * flow(p,r)

! Limits on plant capacity
 forall(p in PLANT) sum(r in REGION) flow(p,r) <= PLANTCAP(p)

! Satisfy all demands
 forall(r in REGION) sum(p in PLANT) flow(p,r) = DEMAND(r)

! Bounds on flows
 forall(p in PLANT, r in REGION | exists(flow(p,r)))
  flow(p,r) <= TRANSCAP(p,r)
```

```
 minimize(MinCost)                        ! Solve the problem

end-model
```

REGION and PLANT are declared to be sets of strings, as yet of unknown size. The data arrays (DEMAND, PLANTCAP, PLANTCOST, TRANSCAP, and DISTANCE) and the array of variables flow are indexed by members of REGION and PLANT, their size is therefore not known at their declaration. The model shows two forms of such array declarations: (1) the arrays DEMAND, PLANTCAP, PLANTCOST are *dense arrays* that are not fixed (all entries corresponding to their index sets exist, new entries are added via assignment or if their index sets grow), (2) the arrays TRANSCAP, DISTANCE), and flow are marked as dynamic, that is, only explicitly assigned or created entries exist — we want to make use of this property in the formulation of the model.

There is a slight difference between dynamic arrays of data and of decision variables (type mpvar): an entry of a data array is created automatically when it is used in the Mosel program, entries of decision variable arrays need to be created explicitly (see Section 3.3.1 below).

The data file transprt.dat contains the problem specific data. It might have, for instance,

```
DEMAND: [ (Scotland) 2840 (North) 2800 (SWest) 2600 (SEast) 2820 (Midlands) 2750]

                     ! [CAP  COST]
PLANTDATA: [ (Corby)   [3000 1700]
             (Deeside) [2700 1600]
             (Glasgow) [4500 2000]
             (Oxford)  [4000 2100] ]

                           ! [DIST CAP]
ROUTES: [ (Corby   North)    [400 1000]
          (Corby   SWest)    [400 1000]
          (Corby   SEast)    [300 1000]
          (Corby   Midlands) [100 2000]
          (Deeside Scotland) [500 1000]
          (Deeside North)    [200 2000]
          (Deeside SWest)    [200 1000]
          (Deeside SEast)    [200 1000]
          (Deeside Midlands) [400  300]
          (Glasgow Scotland) [200 3000]
          (Glasgow North)    [400 2000]
          (Glasgow SWest)    [500 1000]
          (Glasgow SEast)    [900  200]
          (Oxford  Scotland) [800    *]
          (Oxford  North)    [600 2000]
          (Oxford  SWest)    [300 2000]
          (Oxford  SEast)    [200 2000]
          (Oxford  Midlands) [400  500] ]

FUELCOST: 17
```

where we give the ROUTES data only for possible plant/region routes, indexed by the plant and region. It is possible that some data are not specified; for instance, there is no Corby – Scotland route. So the data are *sparse* and we just create the flow variables for the routes that exist. (The '*' for the (Oxford,Scotland) entry in the capacity column indicates that the entry does not exist; we may write '0' instead: in this case the corresponding flow variable will be created but bounded to be 0 by the transport capacity limit).

The *condition* whether an entry in a data table is defined is tested with the Mosel function exists. With the help of the '|' operator we add this test to the forall loop creating the variables. It is not required to add this test to the sums over these variables: only the $\mathrm{flow}_{\mathrm{pr}}$ variables that have been created are taken into account. However, if the sums involve exactly the index sets that have been used in the declaration of the variables (here this is the case for the objective function MinCost), adding the existence test helps to speed up the enumeration of the existing index-tuples. The following section introduces the conditional generation in a more systematic way.

3.3 Conditional generation — the | operator

Suppose we wish to apply an upper bound to some but not all members of a set of variables x_i. There are $MAXI$ members of the set. The upper bound to be applied to x_i is U_i, but it is only to be applied if the entry in the data table TAB_i is greater than 20. If the bound did not depend on the value in TAB_i then the statement would read:

```
forall(i in 1..MAXI) x(i) <= U(i)
```

Requiring the condition leads us to write

```
forall(i in 1..MAXI | TAB(i) > 20 ) x(i) <= U(i)
```

The symbol '|' can be read as 'such that' or 'subject to'.

Now suppose that we wish to model the following

$$\sum_{\substack{i=1 \\ A_i>20}}^{MAXI} x_i \leqslant 15$$

In other words, we just want to include in a sum those x_i for which A_i is greater than 20. This is accomplished by

```
CC:= sum((i in 1..MAXI | A(i)>20 ) x(i) <= 15
```

3.3.1 Conditional variable creation and `create`

As we have already seen in the transport example (Section 3.2), with Mosel we can conditionally create variables. In this section we show a few more examples.

Suppose that we have a set of decision variables `x(i)` where we do not know the set of `i` for which `x(i)` exist until we have read data into a set `WHICH`.

```
model doesx
 public declarations
  IR = 1..15
  WHICH: set of integer
  x: dynamic array(IR) of mpvar
  Obj,C: linctr
 end-declarations

! Read data from file
 initializations from 'doesx.dat'
  WHICH
 end-initializations

! Create the x variables that exist
 forall(i in WHICH) create(x(i))

! Build a little model to show what exists
 Obj:= sum(i in IR) x(i)
 C:= sum(i in IR) i * x(i) >= 5

! Display the resulting problem definition in Mosel
 exportprob("", Obj)
end-model
```

If the data in `doesx.dat` are

```
WHICH: [1 4 7 11 14]
```

the output from the model is

```
Minimize
 x(1) + x(4) + x(7) + x(11) + x(14)
Subject To
C: x(1) + 4 x(4) + 7 x(7) + 11 x(11) + 14 x(14) >= 5
Bounds
End
```

Note: `exportprob("", Obj)` is a nice idiom for seeing on-screen the problem that has been created in Mosel. The `exportprob` routine outputs (the portion of) the problem definition held in Mosel core, ignoring any solver-specific extensions—to include the latter use module-specific output routines such as `writeprob` of *mmxprs* for Xpress Optimizer. The `public` declaration of decision variables and constraints ensures that the display employs the entity names from the model, by default it will only show automatically generated names.

The key point is that `x` has been declared as a *dynamic array*, and then the variables that exist have been created explicitly with `create`.

When we later take operations over the index set of `x` (for instance, summing), we only include those `x` that have been created. Note that with larger data sets it is recommended to add an explicit `exists` condition in such loops to improve performance for the enumeration of sparse arrays (this does not alter the problem definition):

```
 Obj:= sum(i in IR | exists(x(i))) x(i)
 C:= sum(i in IR | exists(x(i))) i * x(i) >= 5
```

Another way to do this, is

```
model doesx2
 public declarations
  WHICH: set of integer
  Obj,C: linctr
 end-declarations

! Read data from file
 initializations from 'doesx.dat'
  WHICH
 end-initializations

 public declarations
  x: array(WHICH) of mpvar        ! Here the array is _not_ dynamic
 end-declarations                 !  because the set has been finalized

! Build a little model to show what exists
 Obj:= sum(i in WHICH) x(i)
 C:= sum(i in WHICH) i * x(i) >= 5

! Display the resulting problem definition in Mosel
 exportprob(0, "", Obj)
end-model
```

By default, an array is of fixed size if all of its indexing sets are of fixed size (*i.e.* they are either constant or have been *finalized*). When initializing the set `WHICH` the *automatic finalization* mechanism of Mosel gets applied. Finalizing turns a dynamic set into a constant set consisting of the elements that are currently in the set. All subsequently declared arrays that are indexed by this set will be created as *static* (= fixed size).

The second method has two advantages: it is more efficient, and it does not require us to think of the limits of the range `IR` *a priori*.

3.4 Reading sparse data

Suppose we want to read in data of the form

$$i, j, \text{value}_{ij}$$

from an ASCII file, setting up a dynamic array `A(range, range)` with just the `A(i,j)= value`$_{ij}$ for the pairs `(i,j)` which exist in the file. Here is an example which shows three different ways of doing this. We read data from differently formatted files into three different arrays, and using `writeln` show that the arrays hold identical data.

3.4.1 Data input with `initializations from`

The first method, using the `initializations` block, has already been introduced (transport problem in Section 3.2).

```
model "Trio input (1)"
 declarations
  A1: dynamic array(range,range) of real
 end-declarations

! First method: use an initializations block
 initializations from 'data_1.dat'
  A1 as 'MYDATA'
 end-initializations

! Now let us see what we have
 writeln('A1 is: ', A1)
end-model
```

The data file `data_1.dat` could be set up thus (every data item is preceded by its index-tuple):

```
MYDATA: [ (1 1) 12.5 (2 3) 5.6 (10 9) -7.1 (3 2) 1 ]
```

This model produces the following output:

```
A1 is: [(1,1,12.5),(2,3,5.6),(3,2,1),(10,9,-7.1)]
```

3.4.2 Data input with `readln`

The second way of setting up and accessing data demonstrates the immense flexibility of `readln`. The format of the data file may be freely defined by the user. After every call to `read` or `readln` the parameter `nbread` contains the number of items read. Its value should be tested to check whether the end of the data file has been reached or an error has occurred (*e.g.* unrecognized data items due to incorrect formating of a data line). Notice that `read` and `readln`interpret spaces as separators between data items; strings containing spaces must therefore be quoted using either single or double quotes.

```
model "Trio input (2)"
 declarations
  A2: dynamic array(range,range) of real
  i, j: integer
 end-declarations

! Second method: use the built-in readln function
 fopen("data_2.dat",F_INPUT)
 repeat
  readln('Tut(', i, 'and', j, ')=', A2(i,j))
 until getparam("nbread") < 6
```

```
 fclose(F_INPUT)

! Now let us see what we have
 writeln('A2 is: ', A2)
end-model
```

The data file `data_2.dat` could be set up thus:

File `data_2.dat`:

```
Tut(1 and 1)=12.5
Tut(2 and 3)=5.6
Tut(10 and 9)=-7.1
Tut(3 and 2)=1
```

When running this second model version we get the same output as before:

```
A2 is: [(1,1,12.5),(2,3,5.6),(3,2,1),(10,9,-7.1)]
```

3.4.3 Data input with `diskdata`

As a third possibility, one may use the `diskdata` I/O driver from module `mmetc` to read in comma separated value (CSV) files. With this driver the data file may contain single line comments preceded with `!`.

```
model "Trio input (3)"
 uses "mmetc"                      ! Required for diskdata

 declarations
  A3: dynamic array(range,range) of real
 end-declarations

! Third method: use diskdata driver
 initializations from 'mmetc.diskdata:'
  A3 as 'sparse,data_3.dat'
 end-initializations

! Now let us see what we have
 writeln('A3 is: ', A3)
end-model
```

The data file `data_3.dat` is set up thus (one data item per line, preceded by its indices, all separated by commas; strings should be quoted using either single or double quotes):

```
1, 1, 12.5
2, 3,  5.6
10,9, -7.1
3, 2, 1
```

We obtain again the same output as before when running this model version:

```
A3 is: [(1,1,12.5),(2,3,5.6),(3,2,1),(10,9,-7.1)]
```

Note: the `diskdata` format is deprecated, it is provided to enable the use of data sets designed for mp-model and does not support certain new features introduced by Mosel.

3.5 I/O error handling

Mosel's default behaviour on encountering an error is to output an error message and exit from model execution. If a model is embedded into an application this behaviour might not always be desirable, particularly in the case of I/O errors. Data filenames (and contents) most often are changed at runtime

and they are therefore relatively more error-prone than invariable parts of the application.

The following modified extract of the 'transport' example from Section 3.2 shows how to implement custom I/O error handling in a Mosel model. To override the default error handling, this example uses `getparam` and `setparam` to access and change the settings of several Mosel parameters:

`ioctrl` Enable/disable user I/O handling. If disabled (default), the model stops when an I/O error has occurred.

`readcnt` Enable/disable counting of entries per label in 'initializations' blocks. Needs to be enabled when using function `getreadcnt`.

`nbread` Number of items recognized by the last `read` procedure or read in by the last 'initializations' block.

`iostatus` Status of the last I/O operation. A non-zero value indicates an error.

`workdir` The current working directory of the model. Data files are searched for relative to the model's working directory—incorrect paths are quite a common source of I/O errors.

Furthermore, we use the function `getfstat` provided by the module *mmsystem* to check whether the data file we are about to access exists and is of a suitable type (regular file).

Model file `readdataerr.mos`:

```
model "I/O error handling"
 uses "mmsystem"

 declarations
  REGION: set of string                ! Set of customer regions
  PLANT: set of string                 ! Set of plants
  DEMAND: array(REGION) of real        ! Demand at regions
  TRANSCAP,DISTANCE: dynamic array(PLANT,REGION) of real   ! Route data
  FUELCOST: real                       ! Fuel cost per unit distance
 end-declarations

 DATAFILE:= 'transprt.dat'

! Check whether the file we want to access exists
 if bittest(getfstat(DATAFILE),SYS_TYP)<>SYS_REG then
  writeln("File '", DATAFILE, "' does not exist or is not a regular file")
  exit(1)
 end-if

 setparam("ioctrl", true)              ! Application handles I/O errors
 setparam("readcnt", true)             ! Enable per label counting

 initializations from DATAFILE
  DEMAND
  [DISTANCE,TRANSCAP] as 'ROUTE'
  FUELCOST
 end-initializations

 if getparam("iostatus") <>0 then      ! Something has gone wrong in last I/O
  writeln("I/O error reading file '", DATAFILE, "'.")
                                       ! Display the working directory
  writeln("Working directory: ", getparam("workdir"))
                                       ! Display total entries read
  writeln("Total number of entries read: ", getparam("nbread"))
                                       ! Check no. of entries read per label
  forall(s in ["DEMAND","ROUTE","FUELCOST"])
   if getreadcnt(s)=0 then
    writeln("No entries read for label '", s, "'.")
   else
    writeln(getreadcnt(s), " entries read for label '", s, "'.")
   end-if
```

```
 end-if

 setparam("ioctrl", false)              ! Revert to default I/O handling
 setparam("readcnt", false)

end-model
```

We have purposely introduced a mistake (the correct label for the route data is 'ROUTES') and running this model therefore displays an error message produced by Mosel, and also the following output produced by our own error reporting.

```
I/O error reading file 'transprt.dat':
Mosel: E-33: Initialization from file  `transprt.dat' failed for:  `ROUTE'.
Working directory: c:/xpress/examples/mosel/UG/A3
Total number of entries read: 6
5 entries read for label 'DEMAND'.
No entries read for label 'ROUTE'.
1 entries read for label 'FUELCOST'.
```

Given that this model implements its own error handling, we might want to entirely disable the display of error messages from Mosel by redirecting the error stream to 'null:', that is, surrounding the 'initializations' block with these lines:

```
 fopen("null:", F_ERROR)                ! Optional: Disable error stream
 ...                                    ! Initialization of data from file
 fclose(F_ERROR)                        ! Stop error redirection
```

Important: always remember to terminate the error stream redirection by closing the selected output file, otherwise you will no longer see any error output from Mosel from the rest of the model.

Instead of completely ignoring the error messages produced by Mosel, we might also choose to save them to a file in order to inspect or display them later on. This may be a physical (text) file, or for example, a text object directly in the model as shown in this code extract:

```
public declarations
 errtxt: text                           ! Text used as file to log errors
end-declarations

fopen("text:errtxt", F_ERROR)           ! Redirect error stream to a file (text)
...                                     ! Initialization of data from file
fclose(F_ERROR)                         ! Stop error redirection

if getparam("iostatus") <>0 then        ! Something has gone wrong in last I/O
 writeln("I/O error reading file '", DATAFILE, "': ", errtxt)
 ...
end-if
```

In the error redirection we have used 'null:' and 'text:', these two are *I/O drivers* which are explained with some more detail in Section 17.1.2. Concerning the type 'text' please see the discussion in Section 17.6.1.

Note: Certain Mosel modules and also the Mosel Libraries have additional functionality for error handling, such as debug settings for ODBC (see the chapter 'mmodbc' of the Mosel Language Reference for details), or the redirection of Mosel streams from applications (as in Sections 13.5 or 14.1.8) of other models (see the example of Section 17.2.3).

CHAPTER 4

Integer Programming

Though many systems can accurately be modeled as Linear Programs, there are situations where discontinuities are at the very core of the decision making problem. There seem to be three major areas where non-linear facilities are required

- where entities must inherently be selected from a discrete set;
- in modeling logical conditions; and
- in finding the global optimum over functions.

Mosel lets you model these non-linearities using a range of discrete entities and then the Xpress Mixed Integer Programming (MIP) optimizer can be used to find the overall (global) optimum of the problem. Usually the underlying structure is that of a Linear Program, but optimization may be used successfully when the non-linearities are separable into functions of just a few variables.

4.1 Integer Programming entities in Mosel

We shall show how to make variables and sets of variables into discrete entities by using the following declarations.

```
declarations
 IR = 1..8                        ! Index range
 WEIGHT: array(IR) of real        ! Weight table
 x: array(IR) of mpvar
end-declarations

WEIGHT:: [ 2, 5, 7, 10, 14, 18, 22, 30]
```

Xpress handles the following discrete entities based on decision variables:

- *Binary variables*: decision variables that can take either the value 0 or the value 1 (do/ don't do variables).
  We make a variable, say `x(4)`, binary by

```
x(4) is_binary
```

- *Integer variables*: decision variables that can take only integer values.
  We make a variable, say `x(7)`, integer by

```
x(7) is_integer
```

- *Partial integer variables*: decision variables that can take integer values up to a specified limit and any value above that limit.

```
x(1) is_partint 5   ! Integer up to 5, then continuous
```

- *Semi-continuous variables*: decision variables that can take either the value 0, or a value between some lower limit and upper limit. Semi-continuous variables help model situations where if a variable is to be used at all, it has to be used at some minimum level.

```
x(2) is_semcont 6   ! A 'hole' between 0 and 6, then continuous
```

- *Semi-continuous integer variables*: decision variables that can take either the value 0, or an integer value between some lower limit and upper limit. Semi-continuous integer variables help model situations where if a variable is to be used at all, it has to be used at some minimum level, and has to be integer.

```
x(3) is_semint 7    ! A 'hole' between 0 and 7, then integer
```

- *Special Ordered Sets of type one (SOS1)*: an ordered set of non-negative variables at most one of which can take a non-zero value.
- *Special Ordered Sets of type two (SOS2)*: an ordered set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering. If the coefficients in the `WEIGHT` array determine the ordering of the variables, we might form a SOS1 or SOS2 set `MYSOS` by

```
MYSOS:= sum(i in IRng) WEIGHT(i)*x(i) is_sosX
```

  where `is_sosX` is either `is_sos1` for SOS1 sets, or `is_sos2` for SOS2 sets.
  Alternatively, if the set `S` holds the members of the set and the linear constraint `L` contains the set variables' coefficients used in ordering the variables (the so-called *reference row entries*), then we can do thus:

```
makesos1(S,L)
```

  with the obvious change for SOS2 sets. This method must be used if the coefficient (here `WEIGHT(i)`) of an intended set member is zero. With `is_sosX` the variable will not appear in the set since it does not appear in the linear expression.
  Another point to note about Special Ordered Sets is that the ordering coefficients must be distinct (or else they are not doing their job of supplying an order!).

The most commonly used entities are *binary variables*, which can be employed to model a whole range of logical conditions. *General integers* are more frequently found where the underlying decision variable really has to take on a whole number value for the optimal solution to make sense. For instance, we might be considering the number of airplanes to charter, where fractions of an airplane are not meaningful and the optimal answer will probably involve so few planes that rounding to the nearest integer may not be satisfactory.

Partial integers provide some computational advantages in problems where it is acceptable to round the LP solution to an integer if the optimal value of a decision variable is quite large, but unacceptable if it is small. *Semi-continuous variables* are useful where, if some variable is to be used, its value must be no less than some minimum amount. If the variable is a *semi-continuous integer variable*, then it has the added restriction that it must be integral too.

Special Ordered Sets of type 1 are often used in modeling choice problems, where we have to select at most one thing from a set of items. The choice may be from such sets as: the time period in which to start a job; one of a finite set of possible sizes for building a factory; which machine type to process a part on. *Special Ordered Sets of type 2* are typically used to model non-linear functions of a variable. They are the natural extension of the concepts of Separable Programming, but when embedded in a

Branch-and-Bound code (see below) enable truly global optima to be found, and not just local optima. (A local optimum is a point where all the nearest neighbors are worse than it, but where we have no guarantee that there is not a better point some way away. A global optimum is a point which we know to be the best. In the Himalayas the summit of K2 is a local maximum height, whereas the summit of Everest is the global maximum height).

The Xpress MIP solver also supports the following special constraint types:

- *Indicator constraints* are logic constraints that associate a binary variable b with a linear inequality constraint C. They take one of these two forms
  'if $\mathrm{b} = 1$ then C', in symbols: $\mathrm{b} \rightarrow \mathrm{C}$, or
  'if $\mathrm{b} = 0$ then C', in symbols: $\mathrm{not}(\mathrm{b}) \rightarrow \mathrm{C}$
  Indicator constraints in Mosel have the type `logctr`, they are defined via the subroutine `indicator` of *mmxprs*, for example

```
declarations
  b: mpvar
  x: array(range) of mpvar
end-declarations
b is_binary
indicator(1, b, x(1)+x(2)>=10)      ! if b>=1 then x(1)+x(2)>=10
```

- *General constraints* are specific constraint relations that are recognized by MIP solvers these include
  - piecewise linear expressions: can be used in place of SOS-2 formulations
  - absolute value, minimum value, maximum value of discrete or continuous decision variables
  - logical constraints: 'and' and 'or' over binary variables

  The Mosel implementation of these constraints comes in the form of the modeling constructs `pwlin`, `fmin`, `fmax`, and `abs`. These constructs define constraints of the type `nlctr` and require the module *mmxnlp* to be loaded, but they will result in MIP models if they only involve linear expressions, such as these constraints (where `x(i)`, `y`, `z` are continuous or discrete decision variables):

```
uses "mmxnlp"
declarations
  x: array(R: range) of mpvar
  y,z: mpvar
end-declarations
abs(x(1)-2*x(2)) <= 10                    ! Absolute value constraint
MinCtr:= fmin(union(i in R) [x(i)]) >= 5  ! Minimum value constraint
y = fmax(x(3), 20, 2*x(1)-z)              ! Maximum value constraint
```

  An example of using piecewise linear expressions is discussed in Section *The purchasing model using piecewise linear expressions* later in this chapter.
  For the formulation of the logical constraints 'and' and 'or' via general constraints Mosel defines a second form of logical constraints in *mmxprs* (like the indicator constraints these are of type `logctr`) based on Boolean variables (type `boolvar`) The resulting constraints are directly sent to the solver as constraints on binary variables as part of the definition of a MIP model, for example:

```
uses "mmxprs"
declarations
  R=1..5
  bv: array(R) of boolvar
  LC: logctr
end-declarations

 bv(1) and not bv(5)      ! Simple clause, same as:  bv(1) and (not bv(5)) = true
 bv(3) = (not bv(4))      ! Association of clauses
```

```
! Defining a logic constraint
 LC:= (or(i in 1..3) not bv(i)) = false

! Solve as feasibility problem (MIP)
 maximize
 if getprobstat=XPRS_OPT then
   forall(i in R) writeln(i, ": ", bv(i).sol)
 end-if
```

Please refer to Section 'Boolean variables' of the 'MIP formulations and linearizations' guide for further detail on the use of logical constraints defined via Boolean variables.

Theoretically, models that can be built with any of the entities we have listed above can be modeled solely with binary variables. The reason why modern IP systems have some or all of the extra entities is that they often provide significant computational savings in computer time and storage when trying to solve the resulting model. Most books and courses on Integer Programming do not emphasize this point adequately. We have found that careful use of the non-binary discrete entities often yields very considerable reductions in solution times over ones that just use binary variables.

To illustrate the use of Mosel in modeling Integer Programming problems, two small examples follow. The first example is initially formulated using binary variables. This formulation is then modified to use Special Ordered Sets of type 1. For the second example we show a formulation via Special Ordered Sets of type 2 and also a simplified representation via 'piecewise linear' constructs.

For the interested reader, an excellent text on Integer Programming is *Integer Programming* by Laurence Wolsey, Wiley Interscience, 1998, ISBN 0-471-28366-5.

4.2 A project planning model

A company has several projects that it must undertake in the next few months. Each project lasts for a given time (its duration) and uses up one resource as soon as it starts. The resource profile is the amount of the resource that is used in the months following the start of the project. For instance, project 1 uses up 3 units of resource in the month it starts, 4 units in its second month, and 2 units in its last month.

The problem is to decide when to start each project, subject to not using more of any resource in a given month than is available. The benefit from the project only starts to accrue when the project has been completed, and then it accrues at BEN_p per month for project p, up to the end of the time horizon. Below, we give a mathematical formulation of the above project planning problem, and then display the Mosel model form.

4.2.1 Model formulation

Let PROJ denote the set of projects and $\mathrm{MONTHS} = \{1, ..., \mathrm{NM}\}$ the set of months to plan for. The data are:

DUR_p	the duration of project p
RESUSE_{pt}	the resource usage of project p in its t^{th} month
RESMAX_m	the resource available in month m
BEN_p	the benefit per month when project finishes

We introduce the binary decision variables start_{pm} that are 1 if project p starts in month m, and 0 otherwise.

The objective function is obtained by noting that the benefit coming from a project only starts to accrue when the project has finished. If it starts in month m then it finishes in month $m + \mathrm{DUR}_p - 1$. So, in total,

we get the benefit of BEN_p for $\mathrm{NM} - (m + \mathrm{DUR}_p - 1) = \mathrm{NM} - m - \mathrm{DUR}_p + 1$ months. We must consider all the possible projects, and all the starting months that let the project finish before the end of the planning period. For the project to complete it must start no later than month $\mathrm{NM} - \mathrm{DUR}_p$. Thus the profit is:

$$\sum_{p \in \mathrm{PROJ}} \sum_{m=1}^{\mathrm{NM}-\mathrm{DUR}_p} (\mathrm{BEN}_p \cdot (\mathrm{NM} - m - \mathrm{DUR}_p + 1)) \cdot \mathrm{start}_{pm}$$

Each project must be done once, so it must start in one of the months 1 to $\mathrm{NM} - \mathrm{DUR}_p$:

$$\forall p \in \mathrm{PROJ} : \sum_{m \in \mathrm{MONTHS}} \mathrm{start}_{pm} = 1$$

We next need to consider the implications of the limited resource availability each month. Note that if a project p starts in month m it is in its $(k - m + 1)^{\mathrm{th}}$ month in month k, and so will be using $\mathrm{RESUSE}_{p,k-m+1}$ units of the resource. Adding this up for all projects and all starting months up to and including the particular month k under consideration gives:

$$\forall k \in \mathrm{MONTHS} : \sum_{p \in \mathrm{PROJ}} \sum_{m=1}^{k} \mathrm{RESUSE}_{p,k+1-m} \cdot \mathrm{start}_{pm} \leqslant \mathrm{RESMAX}_k$$

Finally we have to specify that the start_{pm} are binary (0 or 1) variables:

$$\forall p \in \mathrm{PROJ}, m \in \mathrm{MONTHS} : \mathrm{start}_{pm} \in \{0, 1\}$$

Note that the start month of a project p is given by:

$$\sum_{m=1}^{\mathrm{NM}-\mathrm{DUR}_p} m \cdot \mathrm{start}_{pm}$$

since if an start_{pm} is 1 the summation picks up the corresponding m.

4.2.2 Implementation

The model as specified to Mosel is as follows (file `pplan1.mos`):

```
model Pplan
 uses "mmxprs"

 declarations
  PROJ = 1..3                      ! Set of projects
  NM = 6                           ! Time horizon (months)
  MONTHS = 1..NM                   ! Set of time periods (months) to plan for

  DUR: array(PROJ) of integer      ! Duration of project p
  RESUSE: array(PROJ,MONTHS) of integer
                                   ! Res. usage of proj. p in its t'th month
  RESMAX: array(MONTHS) of integer ! Resource available in month m
  BEN: array(PROJ) of real         ! Benefit per month once project finished

  start: array(PROJ,MONTHS) of mpvar  ! 1 if proj p starts in month t, else 0
 end-declarations

 DUR   :: [3, 3, 4]
 RESMAX:: [5, 6, 7, 7, 6, 6]
 BEN   :: [10.2, 12.3, 11.2]
 RESUSE:: (1,1..3)[3, 4, 2]
 RESUSE:: (2,1..3)[4, 1, 5]
```

```
 RESUSE:: (3,1..4)[3, 2, 1, 2]     ! Other RESUSE entries are 0 by default

! Objective: Maximize Benefit
! If project p starts in month t, it finishes in month t+DUR(p)-1 and
! contributes a benefit of BEN(p) for the remaining NM-(t+DUR(p)-1) months:
 MaxBen:=
  sum(p in PROJ, m in 1..NM-DUR(p)) (BEN(p)*(NM-m-DUR(p)+1)) * start(p,m)

! Each project starts once and only once:
 forall(p in PROJ) One(p):= sum(m in MONTHS) start(p,m) = 1

! Resource availability:
! A project starting in month m is in its k-m+1'st month in month k:
 forall(k in MONTHS) ResMax(k):=
  sum(p in PROJ, m in 1..k) RESUSE(p,k+1-m)*start(p,m) <= RESMAX(k)

! Make all the start variables binary
 forall(p in PROJ, m in MONTHS) start(p,m) is_binary

 maximize(MaxBen)                   ! Solve the MIP-problem

 writeln("Solution value is: ", getobjval)
 forall(p in PROJ) writeln( p, " starts in month ",
                           getsol(sum(m in MONTHS) m*start(p,m)) )
end-model
```

Note that in the solution printout we apply the `getsol` function not to a single variable but to a linear expression.

4.3 The project planning model using Special Ordered Sets

The example can be modified to use Special Ordered Sets of type 1 (SOS1). The $\mathrm{start_{pm}}$ variables for a given p form a set of variables which are ordered by m, the month. The ordering is induced by the coefficients of the $\mathrm{start_{pm}}$ in the specification of the SOS. For example, $\mathrm{start_{p1}}$'s coefficient, 1, is less than $\mathrm{start_{p2}}$'s, 2, which in turn is less than $\mathrm{start_{p3}}$'s coefficient, and so on The fact that the $\mathrm{start_{pm}}$ variables for a given p form a set of variables is specified to Mosel as follows:

```
(! Define SOS-1 sets that ensure that at most one start(p,m) is non-zero
   for each project p. Use month index to order the variables !)

 forall(p in PROJ) XSet(p):= sum(m in MONTHS) m*start(p,m) is_sos1
```

The `is_sos1` specification tells Mosel that `Xset(p)` is a Special Ordered Set of type 1.

The linear expression specifies both the set members and the coefficients that order the set members. It says that all the $\mathrm{start_{pm}}$ variables for m in the MONTHS index range are members of an SOS1 with reference row entries m.

The specification of the $\mathrm{start_{pm}}$ as binary variables must now be removed. The binary nature of the $\mathrm{start_{pm}}$ is implied by the SOS1 property, since if the $\mathrm{start_{pm}}$ must add up to 1 and only one of them can differ from zero, then just one is 1 and the others are 0.

If the two formulations are equivalent why were Special Ordered Sets invented, and why are they useful? The answer lies in the way the reference row gives the search procedure in Integer Programming (IP) good clues as to where the best solution lies. Quite frequently the Linear Programming (LP) problem that is solved as a first approximation to an Integer Program gives an answer where $\mathrm{start_{p1}}$ is fractional, say with a value of 0.5, and $\mathrm{start_{p,NM}}$ takes on the same fractional value. The IP will say:

'my job is to get variables to 0 or 1. Most of the variables are already there so I will try moving xp1 or xpT. Since the set members must add up to 1.0, one of them will go to 1, and one to 0. So I think that we

start the project either in the first month or in the last month.'

A much better guess is to see that the $\mathrm{start}_{\mathrm{pm}}$ are ordered and the LP solution is telling us it looks as if the best month to start is somewhere midway between the first and the last month. When sets are present, the IP can branch on sets of variables. It might well separate the months into those before the middle of the period, and those later. It can then try forcing all the early $\mathrm{start}_{\mathrm{pm}}$ to 0, and restricting the choice of the one $\mathrm{start}_{\mathrm{pm}}$ that can be 1 to the later $\mathrm{start}_{\mathrm{pm}}$. It has this option because it now has the information to 'know' what is an early and what is a late $\mathrm{start}_{\mathrm{pm}}$, whereas these variables were unordered in the binary formulation.

The power of the set formulation can only really be judged by its effectiveness in solving large, difficult problems. When it is incorporated into a good IP system such as Xpress it is often found to be an order of magnitude better than the equivalent binary formulation for large problems.

4.4 Purchasing with price breaks

There are three suppliers of a good, and they have quoted various prices for various quantities of product. We want to buy at least total cost, yet not buy too much from any one supplier. Each supplier offers decreasing prices for increased lot size, in the form of incremental discounts (the data is given in Table 4.1). We wish to buy 600 items in total.

Table 4.1: Price profile

Supplier	Unit price 1	Breakpoint 1	Unit price 2	Breakpoint 2	Unit price 3	Breakpoint 3
1	9.2	100	9	200	7	1000
2	9	50	8.5	250	8.3	2000
3	11	100	8.5	300	7.5	4000

For example, if you buy 150 items from supplier 1, you pay $9.2 \cdot 100 + 9 \cdot 50$. If you buy 225 items from supplier 1, you pay $9.2 \cdot 100 + 9 \cdot 100 + 7 \cdot 25$.

4.4.1 Model formulation

We write SUPPL for the set of suppliers, $\mathrm{BREAK0}$ for the set of breakpoints including 0, and $\mathrm{BREAK1}$ the set of breakpoints starting with the first point where unit cost changes, that is without 0. For every supplier s, B_{sb} ($\mathrm{b} \in \mathrm{BREAK1}$) denotes the breakpoint where the unit cost changes from $\mathrm{COST}_{\mathrm{sb}}$ to $\mathrm{COST}_{\mathrm{s,b+1}}$; B_{s0} is always 0. Furthermore REQ denotes the total amount we wish to buy, and $\mathrm{MAXPERC}_{\mathrm{s}}$ the maximum percentage that may be bought from supplier s.

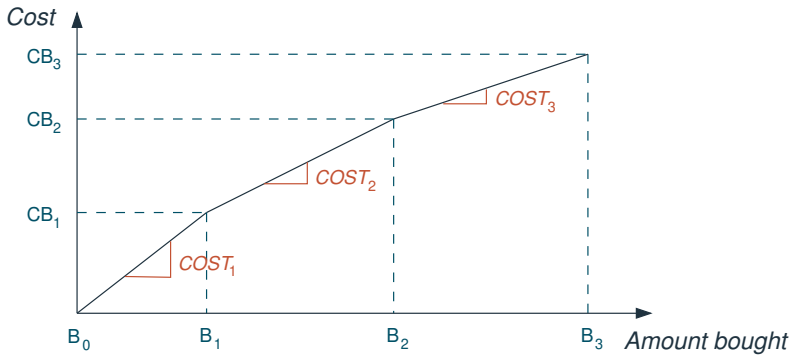


Figure 4.1: Price curve of a supplier

To formulate this problem, we introduce two sets of variables. $\mathrm{buy}_{\mathrm{s}}$ represents the quantity bought

from supplier s. These quantities are calculated via weight variables w_{sb} associated with the breakpoints. The weight variables for every supplier s form an *SOS-2* (*Special Ordered Set of Type 2*), that means that at most two neighboring variables w_{sb} in the order induced by the coefficients BR_{bs} may be non-zero. Under the condition that the weight variables sum up to 1 for every supplier we can use these variables to linearly interpolate between the breakpoints to obtain the quantities bought:

$$\forall s \in \mathrm{SUPPL}: \sum_{b \in \mathrm{BREAK0}} \mathrm{B}_{sb} \cdot w_{sb} = \mathrm{buy}_s$$
$$\forall s \in \mathrm{SUPPL}: \sum_{b \in \mathrm{BREAK0}} w_{sb} = 1$$

Furthermore, we need to express the constraints that the total requirement must be met and that no more than a certain given percentage may be bought of each supplier.

$$\sum_{s \in \mathrm{SUPPL}} \mathrm{buy}_s \geqslant \mathrm{REQ}$$
$$\forall s \in \mathrm{SUPPL}: \mathrm{buy}(s) \leqslant \mathrm{MAXPERC}_s/100 \cdot \mathrm{REQ}$$

The total cost at the breakpoints, CB_{sb}, is given by the following recursive formula:

$$\mathrm{CB}_{s0} = 0$$
$$\forall b \in \mathrm{BREAK1}: \mathrm{CB}_{sb} = \mathrm{CB}_{s,b-1} + \mathrm{COST}_{sb} \cdot (\mathrm{B}_{sb} - \mathrm{B}_{s,b-1})$$

With these total cost values per supplier we can formulate the objective function which is to minimize the total cost of the purchase.

$$\text{minimize} \sum_{s \in \mathrm{SUPPL}} \sum_{b \in \mathrm{BREAK0}} \mathrm{CB}_{sb} \cdot w_{sb}$$

4.4.2 Implementation

The Mosel implementation of the mathematical model looks as follows (file `purch.mos`).

```
model "Purchase"
 uses "mmxprs"

 declarations
   NB = 3
   BREAK1 = 1..NB                     ! Price breaks
   BREAK0 = 0..NB                     ! Price breaks including 0
   SUPPL = 1..3                       ! Suppliers

   COST: array(SUPPL,BREAK1) of real  ! Unit cost
   B: array(SUPPL,BREAK0) of real     ! Breakpoints (quantities at which unit
                                      ! cost changes)
   CB: array(SUPPL,BREAK0) of real    ! Total cost at break points
   MAXPERC: array(SUPPL) of real      ! Maximum percentages for each supplier
   REQ: real                          ! Total quantity required

   buy: array(SUPPL) of mpvar         ! Quantity to purchase from supplier s
   w: array(SUPPL,BREAK0) of mpvar    ! Weights at breakpoint k for supplier s
 end-declarations

 initializations from 'purch.dat'
   COST B MAXPERC REQ
 end-initializations

! Calculate total cost at breakpoints
 forall(s in SUPPL) do
   CB(s,0):= 0
```

```
  forall(b in BREAK1) CB(s,b):= CB(s,b-1) + COST(s,b)*(B(s,b)-B(s,b-1))
 end-do

! Objective: sum of costs*weights
 MinCost:= sum(s in SUPPL, b in BREAK0) CB(s,b) * w(s,b)

! Define buy and also order the weight variables by breakpoint quantities
 forall(s in SUPPL) sum(b in BREAK0) B(s,b) * w(s,b) = buy(s)

! The convexity row (w sum to 1)
 forall(s in SUPPL) sum(b in BREAK0) w(s,b) = 1

! The minimum quantity that must be bought
 sum(s in SUPPL) buy(s) >= REQ

! No more than the maximum percentage from each supplier
 forall(s in SUPPL) buy(s) <= MAXPERC(s)*REQ/100.0

! Define the w as SOS-2
 forall(s in SUPPL)
   makesos2(union(b in BREAK0) {w(s,b)}, sum(b in BREAK0) B(s,b)*w(s,b))

 minimize(MinCost)              ! Solve the MIP-problem

 writeln("Minimum total cost: ", getobjval)
 forall(s in SUPPL) writeln(" buy(",s,"): ", getsol(buy(s)), " cost: ",
   getsol(sum(b in BREAK0) CB(s,b) * w(s,b)) )

end-model
```

In the model above we use the procedure `makesos2` for the definition of the SOS-2. Alternatively, if we wish to use `is_sos2`, then the weight coefficients `B(s,b)` all need to be augmented by some non-zero constant `EPS` since Mosel does not accept 0-valued weights with `is_sos2`.

```
forall(s in SUPPL) sum(b in BREAK0) (B(s,b)+EPS) * w(s,b) is_sos2
```

The minimum cost of € 5,435 is obtained by buying 240 items from supplier 1, 210 from supplier 2, and 150 items from supplier 3.

4.5 The purchasing model using piecewise linear expressions

The Xpress MIP solver supports a special construct for the formulation of *piecewise linear expressions* like the price curves of individual suppliers in the purchase problem that we have seen in the previous section. With this construct the modeler no longer needs to define the weight variables associated with the break points of piecewise linear expressions, these are handled internally by the solver.

With Mosel there are three different ways of specifying piecewise linear expressions, namely via

1. *lists of points* (the coordinates $(\mathrm{B}_{\mathrm{sb}},\mathrm{CB}_{\mathrm{sb}})$ of cost breakpoints per supplier s in our example); the points must be given in increasing order of X-axis values, if all points are different they represent a continuous function; discontinuous functions can be modeled by duplicating a point;
2. *slopes* (in our case slopes are given by the unit cost values $\mathrm{COST}_{\mathrm{sb}}$); this form defines a continuous function with its starting point at the origin of the coordinate system, that is the point (0,0);
3. *linear segments* (a list of linear expressions, in our example a function of the cumulative cost $\mathrm{CB}_{\mathrm{sb}}$ at the start of the segment and the slope $\mathrm{COST}_{\mathrm{sb}}$); this form can be used to represent discontinuous piecewise linear functions where there are 'jumps' at the breakpoints.

The supplier price functions in the purchase problem are continuous with start at 0, so all three forms can be used. The following Mosel implementation (file `purch_pwl.mos`) uses the first formulation variant for the `pwlin` construct. In place of the binary weight variables there now is a single variable $\texttt{pay}_s$ per supplier for the cost incurred from this supplier that is used for the formulation of the objective function, all else remains as before. It should be noted that the solver is started via the Xpress NLP module (see Section 17.4) that handles the transformation of the `pwlin` construct into matrix form, however, this problem will be solved as a MIP.

```
model "Purchase pwl"
 uses "mmxnlp"

 declarations
   NB = 3
   BREAK1 = 1..NB                     ! Price breaks
   BREAK0 = 0..NB                     ! Price breaks including 0
   SUPPL = 1..3                       ! Suppliers

   COST: array(SUPPL,BREAK1) of real  ! Unit cost
   B: array(SUPPL,BREAK0) of real     ! Breakpoints (quantities at which unit
                                      ! cost changes)
   CB: array(SUPPL,BREAK0) of real    ! Total cost at break points
   MAXPERC: array(SUPPL) of real      ! Maximum percentages for each supplier
   REQ: real                          ! Total quantity required

   buy: array(SUPPL) of mpvar         ! Quantity to purchase from supplier s
   pay: array(SUPPL) of mpvar         ! Cost incurred from supplier s
 end-declarations

 initializations from 'purchase.dat'
   COST B MAXPERC REQ
 end-initializations

! Calculate total cost at breakpoints
 forall(s in SUPPL) do
   CB(s,0):= 0
   forall(b in BREAK1) CB(s,b):= CB(s,b-1) + COST(s,b)*(B(s,b)-B(s,b-1))
 end-do

! Objective: sum of costs*weights
 MinCost:= sum(s in SUPPL) pay(s)

! Define relation between bought quantities and price paid per supplier
 forall(s in SUPPL)
 ! Formulation of pwlin via breakpoint coordinates
   pay(s) = pwlin(buy(s), union(b in BREAK0) [B(s,b),CB(s,b)])

! The minimum quantity that must be bought
 Demand:= sum(s in SUPPL) buy(s) >= REQ

! No more than the maximum percentage from each supplier
 forall(s in SUPPL) buy(s) <= MAXPERC(s)*REQ/100.0

 minimize(MinCost)              ! Solve the MIP-problem

 writeln("Minimum total cost: ", getobjval)
 forall(s in SUPPL) writeln(" buy(",s,"): ", getsol(buy(s)), " cost: ", pay(s).sol)

end-model
```

The alternative forms of the `pwlin` specification are

```
! Formulation of pwlin via slopes
  pay(s) = pwlin(buy(s), union(b in 1..(NB-1)) [B(s,b)], union(b in BREAK1) [COST(s,b)])
```

and

```
! Formulation of pwlin via segments
  pay(s) = pwlin(union(b in 0..(NB-1)) [pws(B(s,b),CB(s,b)+COST(s,b+1)*(buy(s)-B(s,b)))])
```

where `pws` states a *segment of a piecewise linear function*.

CHAPTER 5

Overview of subroutines and reserved words

There is a range of built-in functions and procedures available in Mosel. They are described fully in the Mosel Language Reference Manual. Here is a summary.

- Accessing solution values: `getsol`, `getact`, `getdual`, `getrcost`, `getslack`, `getobjval`
- Arithmetic functions: `abs`, `arctan`, `cos`, `sin`, `ceil`, `floor`, `round`, `exp`, `ln`, `log`, `sqrt`, `isodd`, `random`, `setrandseed`
- List functions: `maxlist`, `minlist`, `cutelt`, `cutfirst`, `cutlast`, `cuthead`, `cuttail`, `findfirst`, `findlast`, `getelt`, `getfirst`, `getlast`, `getreverse`, `reverse`, `gethead`, `gettail`, `splithead`, `splittail`
- String functions: `strfmt`, `substr`, `_`
- Dynamic array handling: `create`, `exists`, `finalize`, `delcell`, `isdynamic`
- File handling: `fclose`, `fflush`, `fopen`, `fselect`, `fskipline`, `fwrite`, `fwrite_`, `fwriteln`, `fwriteln_`, `getfid`, `getfname`, `getreadcnt`, `iseof`, `read`, `readln`, `write`, `write_`, `writeln`, `writeln_`
- Accessing control parameters: `getparam`, `localsetparam`, `restoreparam`, `setparam`
- Getting information: `getcoeff`, `getcoeffs`, `getnbdim`, `getsize`, `gettype`, `getvars`
- Constraint definition: `sethidden`, `ishidden`, `makesos1`, `makesos2`, `setcoeff`, `setname`, `setrange`, `settype`
- Time and date: `currentdate`, `currenttime`, `timestamp`
- Bit values: `bitflip`, `bitneg`, `bitset`, `bitshift`, `bittest`, `bitval`
- Special values: `isfinite`, `isinf`, `isnan`
- Handling unions: `geteltype`, `getstruct`, `gettypeid`, `isdefined`
- Miscellaneous functions: `asproc`, `assert`, `compare`, `datablock`, `dumpcallstack`, `exit`, `exportprob`, `reset`, `setioerr`, `setmatherr`, `publish`, `unpublish`, `memoryuse`, `newmuid`, `versionnum`, `versionstr`

5.1 Modules

The distribution of Mosel contains several *modules* that add extra functionality to the language.

A full list of the functionality of a module can be obtained by using Mosel's `exam` command, for instance

```
mosel exam mmsystem
```

In this manual, we always use Xpress Optimizer as solver. Access to the corresponding optimization functions is provided by the module `mmxprs`.

In the `mmxprs` module are the following useful functions.

- Optimize: `minimize`, `maximize`
- MIP directives: `setmipdir`, `clearmipdir`
- Handling bases: `savebasis`, `loadbasis`, `delbasis`
- Force problem loading: `loadprob`
- Accessing problem status: `getprobstat`
- Deal with bounds: `setlb`, `setub`, `getlb`, `getub`
- Model cut functions: `setmodcut`, `clearmodcut`

For example, here is a nice habit to get into when solving a problem with Xpress Optimizer.

```
declarations
  status:array({XPRS_OPT,XPRS_UNF,XPRS_INF,XPRS_UNB,XPRS_OTH}) of string
end-declarations

status::([XPRS_OPT,XPRS_UNF,XPRS_INF,XPRS_UNB,XPRS_OTH])[
          "Optimum found","Unfinished","Infeasible","Unbounded","Failed"]
...
minimize(Obj)
writeln(status(getprobstat))
```

In the `mmsystem` module are various useful functions provided by the underlying operating system and a few programming utilities :

- Delete a file/directory: `fdelete`, `removedir`
- Copy/move a file: `fcopy`, `fmove`
- Make a directory: `makedir`
- Current working directory: `getcwd`
- Get/set an environment variable's value: `getenv`, `setenv`
- File and system status: `getfstat`, `getsysstat`
- General system call: `system`
- Time and date: `gettime`, `getdate`, `getweekday`, `getasnumber`, ...
- Handling the type `text`: `copytext`, `cuttext`, `deltext`, `readtextline`, ...
- Sort an array of any type with 'order' property: `qsort`

Other modules mentioned in this manual are `mmodbc`, `mmsheet`, `mmetc`, and `mmjobs`.

See the module documentation in the *Mosel Language Reference Manual* or in the individual module reference manuals for full details.

5.2 Reserved words

The following words are reserved in Mosel. The upper case versions are also reserved (*i.e.* `AND` and `and` are keywords but not `And`). Do not use them in a model except with their built-in meaning.

`and`, `any`, `array`, `as`
`boolean`, `break`
`case`, `constant`, `count`, `counter`
`declarations`, `div`, `do`, `dynamic`
`elif`, `else`, `end`, `evaluation`
`false`, `forall`, `forward`, `from`, `function`
`hashmap`
`if`, `imports`, `in`, `include`, `initialisations`, `initializations`, `integer`, `inter`, `is`, `is_binary`, `is_continuous`, `is_free`, `is_integer`, `is_partint`, `is_semcont`, `is_semint`, `is_sos1`, `is_sos2`
`linctr`, `list`
`max`, `min`, `mod`, `model`, `mpproblem`, `mpvar`
`namespace`, `next`, `not`, `nsgroup`, `nssearch`
`of`, `options`, `or`
`package`, `parameters`, `procedure`, `public`, `prod`
`range`, `real`, `record`, `repeat`, `requirements`, `return`
`set`, `shared`, `string`, `sum`
`then`, `to`, `true`
`union`, `until`, `uses`
`version`
`while`, `with`

CHAPTER 6

Correcting errors in Mosel models

The parser of Mosel is able to detect a large number of errors that may occur when writing a model. In this chapter we shall try to analyze and correct some of these. As a next step, we also show how to obtain information for dealing with run time errors.

Other types of errors that are in general more difficult to detect are mistakes in the data or logical errors in the formulation of Mosel models—you may use the Mosel Debugger (see Section 15.1) to trace these.

6.1 Correcting syntax errors in Mosel

If we compile the model `poerror1.mos`

```
model  `Plenty of errors'
 declarations
  small, large: mpvar
 end-declarations

 Profit=   5*small + 20*large
 Boxwood:=   small + 3*large <= 200
 Lathe:=   3*small + 2*large <= 160

 maximize(Profit)

 writeln("Best profit is ", getobjval
end-model
```

we get the following output:

```
Mosel: E-100 at (1,7) of  `poerror.mos': Syntax error before  ` `'.
Parsing failed.
```

The second line of the output informs us that the compilation has not been executed correctly. The first line tells us exactly the type of the error that has been detected, namely a syntax error with the code `E-100` (where `E` stands for error) and its location: line 1 character 7. The problem is caused by the apostrophe ` (or something preceding it). Indeed, Mosel expects either single or double quotes around the name of the model if the name contains blanks. We therefore replace it by ' and compile the corrected model, resulting in the following display:

```
Mosel: E-100 at (6,8) of  `poerror.mos': Syntax error before  `='.
Mosel: W-121 at (6,29) of  `poerror.mos': Statement with no effect.
Mosel: E-100 at (10,16) of  `poerror.mos':  `Profit' is not defined.
Mosel: E-123 at (10,17) of  `poerror.mos':  `maximize' is not defined.
Mosel: E-100 at (12,37) of  `poerror.mos': Syntax error.
Parsing failed.
```

There is a problem with the sign = in the 6^{th} line:

```
Profit=  5*small + 20*large
```

In the model body the equality sign = may only be used in the definition of constraints or in logical expressions. Constraints are linear relations between variables, but `profit` has not been defined as a variable, so the parser detects an error. What we really want, is to assign the linear expression `5*small + 20*large` to `Profit`. For such an assignment we have to use the sign `:=`. Using just = is a very common error.

As a consequence of this error, the linear expression after the equality sign does not have any relevance to the problem that is stated. The parser informs us about this fact in the second line: it has found a statement with no effect. This is not an error that would cause the failure of the compilation taken on its own, but simply a *warning* (marked by the `W` in the error code `W-121`) that there may be something to look into. Since `Profit` has not been defined, it cannot be used in the call to the optimization, hence the third error message.

As we have seen, the second and the third error messages are consequences of the first mistake we have made. Before looking at the last message that has been displayed we recompile the model with the corrected line

```
Profit:=  5*small + 20*large
```

to get rid of all side effects of this error. Unfortunately, we still get a few error messages:

```
Mosel: E-123 at (10,17) of  `poerror.mos':  `maximize' is not defined.
Mosel: E-100 at (12,37) of  `poerror.mos': Syntax error.
```

There is still a problem in line 10; this time it shows up at the very end of the line. Although everything appears to be correct, the parser does not seem to know what to do with `maximize`. The solution to this enigma is that we have forgotten to load the module `mmxprs` that provides the optimization function `maximize`. To tell Mosel that this module is used we need to add the line

```
uses "mmxprs"
```

immediately after the start of the model, before the declarations block. Forgetting to specify `mmxprs` is another common error. We now have a closer look at line 12 (which has now become line 13 due to the addition of the `uses` statement). All subroutines called in this line (`writeln` and `getobjval`) are provided by Mosel, so there must be yet another problem: we have forgotten to close the parentheses. After adding the closing parenthesis after `getobjval` the model finally compiles without displaying any errors. If we run it we obtain the desired output:

```
Best profit is 1333.33
Returned value: 0
```

6.2 Correcting run time errors in Mosel

Besides the detection of syntax errors, Mosel may also give some help in finding run time errors. It should only be pointed out here that it is possible to add the flag `-g` to the compile command to obtain some information about where the error occurred in the program, resulting in a command sequence such as

```
mosel compile -g mymodel.mos
mosel mymodel.bim
```

or short

```
mosel exec -g mymodel
```

Also useful is turning on verbose reporting, for instance

```
setparam("XPRS_VERBOSE",true)
setparam("XPRS_LOADNAMES",true)
```

II. Advanced language features

Overview

This part takes the reader who wants to use Mosel as a modeling, solving *and* programming environment through its powerful programming language facilities. The following topics, most of which have already shortly been mentioned in the first part, are covered in a more detailed way:

- Selections and loops (Chapter 7)
- Working with arrays, sets, lists, and records, defining user types and unions (Chapter 8)
- Functions and procedures (Chapter 9)
- Output to files and producing formatted output (Chapter 10)

Whilst the first four chapters in this part present pure programming examples, the last two chapters contain some advanced examples of LP and MIP that make use of the programming facilities in Mosel:

- Cut generation (Section 11.1)
- Column generation (Section 11.2)
- Recursion or Successive Linear Pogramming (Section 12.1)
- Goal Programming (Section 12.2)

CHAPTER 7

Flow control constructs

Flow control constructs are mechanisms for controlling the order of the execution of the actions in a program. In this chapter we are going to have a closer look at two fundamental types of control constructs in Mosel: selections and loops.

Frequently actions in a program need to be repeated a certain number of times, for instance for all possible values of some index or depending on whether a condition is fulfilled or not. This is the purpose of *loops*. Since in practical applications loops are often interwoven with conditions (*selection statements*), these are introduced first.

7.1 Selections

Mosel provides several statements to express a selection between different actions to be taken in a program. The simplest form of a selection is the `if` statement:

- **if**: 'If a condition holds do something'. For example:

```
if A >= 20: x <= 7
```

  For an integer number `A` and a variable `x` of type `mpvar`, `x` is constrained to be less or equal to 7 if `A` is greater or equal 20.

Note that the `if` statement can only be followed by a single expression. With the `if-then` statement there may be any number of expressions between `then` and `end-if`, not just a single one:

- **if-then**: 'If a condition holds do something' (possibly several actions).

```
if A >= 20 then
  x <= 7
end-if
```

In other cases, it may be necessary to express choices with alternatives.

- **if-then-else**: 'If a condition holds, do this, otherwise do something else'. For example:

```
if A >= 20 then
  x <= 7
else x >= 35
end-if
```

  Here the upper bound 7 is applied to the variable `x` if the value of `A` is greater or equal 20, otherwise the lower bound 35 is applied to it.

- **if-then-elif-then-else**: 'If a condition holds do this, otherwise, if a second condition holds do something else *etc.*'

```
if A >= 20 then
  x <= 7
elif A <= 10 then
  x >= 35
else
  x = 0
end-if
```

  Here the upper bound 7 is applied to the variable `x` if the value of `A` is greater or equal 20, and if the value of `A` is less or equal 10 then the lower bound 35 is applied to `x`. In all other cases (that is, `A` is greater than 10 and smaller than 20), `x` is fixed to 0.
  Note that this could also be written using two separate `if-then` statements but it is more efficient to use `if-then-elif-then[-else]` if the cases that are tested are mutually exclusive.

- **case**: 'Depending on the value of an expression do something'.

```
case A of
  -MAX_INT..10 : x >= 35
  20..MAX_INT :  x <= 7
  12, 15 :       x = 1
  else           x = 0
end-case
```

  Here the upper bound 7 is applied to the variable `x` if the value of `A` is greater or equal 20, and the lower bound 35 is applied if the value of `A` is less or equal 10. In addition, `x` is fixed to 1 if `A` has value 12 or 15, and fixed to 0 for all remaining values.
  An example for the use of the case statement is given in Section 12.2.

The following example (model `minmax.mos`) uses the `if-then-elif-then` statement to compute the minimum and the maximum of a set of randomly generated numbers:

```
model Minmax

 declarations
   SNumbers: set of integer
   LB=-1000                    ! Elements of SNumbers must be between LB
   UB=1000                     ! and UB
 end-declarations

! Generate a set of 50 randomly chosen numbers
 forall(i in 1..50)
   SNumbers += {round(random*200)-100}

 writeln("Set: ", SNumbers, " (size: ", getsize(SNumbers), ")")

 minval:=UB
 maxval:=LB
 forall(p in SNumbers)
   if p<minval then
     minval:=p
   elif p>maxval then
     maxval:=p
   end-if

 writeln("Min: ", minval, ", Max: ", maxval)

end-model
```

Instead of writing the loop above, it would of course be possible to use the corresponding operators `min` and `max` provided by Mosel:

```
  writeln("Min: ", min(p in SNumbers) p, ", Max: ", max(p in SNumbers) p)
```

It is good programming practice to indent the block of statements in loops or selections as in the preceding example so that it becomes easy to get an overview where the loop or the selection ends. — At the same time this may serve as a control whether the loop or selection has been terminated correctly (*i.e.* no `end-if` or similar key words terminating loops have been left out).

7.2 Loops

Loops group actions that need to be repeated a certain number of times, either for all values of some index or counter (`forall`) or depending on whether a condition is fulfilled or not (`while`, `repeat-until`).

This section presents the complete set of loops available in Mosel, namely `forall`, `forall-do`, `with`, `while`, `while-do`, and `repeat-until`.

7.2.1 `forall`

The `forall` loop repeats a statement or block of statements for all values of an index or counter. If the set of values is given as an interval of integers (`range`), the enumeration starts with the smallest value. For any other type of sets the order of enumeration depends on the current (internal) order of the elements in the set.

The `forall` loop exists in two different versions in Mosel. The inline version of the `forall` loop (*i.e.* looping over a single statement) has already been used repeatedly, for example as in the following loop that constrains variables `x(i)` (i=1,...,10) to be binary.

```
  forall(i in 1..10) x(i) is_binary
```

The second version of this loop, `forall-do`, may enclose a block of statements, the end of which is marked by `end-do`.

Note that the indices of a `forall` loop can *not* be modified inside the loop. Furthermore, they must be new objects: a symbol that has been declared cannot be used as index of a `forall` loop.

The following example (model `perfect.mos`) that calculates all perfect numbers between 1 and a given upper limit combines both types of the `forall` loop. (A number is called *perfect* if the sum of its divisors is equal to the number itself.)

```
model Perfect

 parameters
  LIMIT=100
 end-parameters

 writeln("Perfect numbers between 1 and ", LIMIT, ":")

 forall(p in 1..LIMIT) do
   sumd:=1
   forall(d in 2..p-1)
    if p mod d = 0 then          ! Mosel's built-in mod operator
      sumd+=d                    ! The same as sum:= sum + d
     end-if
   if p=sumd then
     writeln(p)
   end-if
 end-do

end-model
```

The outer loop encloses several statements, we therefore need to use `forall-do`. The inner loop only applies to a single statement (`if` statement) so that we may use the inline version `forall`.

If run with the default parameter settings, this program computes the solution 1, 6, 28.

7.2.1.1 Multiple indices

The `forall` statement (just like the `sum` operator and any other statement in Mosel that requires index set(s)) may take any number of indices, with values in sets of any basic type or ranges of integer values. If two or more indices have the same set of values as in

```
forall(i in 1..10, j in 1..10) y(i,j) is_binary
```

(where `y(i,j)` are variables of type `mpvar`) the following equivalent short form may be used:

```
forall(i,j in 1..10) y(i,j) is_binary
```

7.2.1.2 Conditional looping

The possibility of adding conditions to a `forall` loop via the '|' symbol has already been mentioned in Chapter 3. Conditions may be applied to one or several indices and the selection statement(s) can be placed accordingly. Take a look at the following example where `A` and `U` are one- and two-dimensional arrays of integers or reals respectively, and `y` a two-dimensional array of decision variables (`mpvar`):

```
forall(i in -10..10, j in 0..5 | A(i) > 20) y(i,j) <= U(i,j)
```

For all `i` from -10 to 10, the upper bound `U(i,j)` is applied to the variable `y(i,j)` if the value of `A(i)` is greater than 20.

The same conditional loop may be reformulated (in an equivalent but usually less efficient way) using the `if` statement:

```
forall(i in -10..10, j in 0..5)
 if A(i) > 20
  y(i,j) <= U(i,j)
end-if
```

If we have a second selection statement on both indices with `B` a two-dimensional array of integers or reals, we may either write

```
forall(i in -10..10, j in 0..5 | A(i) > 20 and B(i,j) <> 0 ) y(i,j) <= U(i,j)
```

or, more efficiently, since the second condition on both indices is only tested if the condition on index `i` holds:

```
forall(i in -10..10 | A(i) > 20, j in 0..5 | B(i,j) <> 0 ) y(i,j) <= U(i,j)
```

7.2.1.3 Counters

A recurring programming task when working with loops, and in particular with conditional loops, is to determine the number of times that the loop has been executed (or that the loop condition is veryfied). To this aim, Mosel provides the construct `as counter`, to be added to the indices of `forall` loops or other statements involving indices, such as `sum`, `max`, or `union` statements.

The following example (see file `count1.mos`) counts and displays all strings in a list that contain the substring `'b'`:

```
L:= ['a', 'ab', 'abc', 'da', 'bc', 'db']
scnt:=0
forall(scnt as counter, s in L | findtext(s, 'b', 1)>0)
  writeln(scnt, ": ", s)
```

The position of `as counter` among the loop indices is entirely up to the programmer and takes no effect on its value. However notice that loop conditions must succeed an index. So for instance, instead

of the above we might equally have written:

```
forall(s in L | findtext(s, 'b', 1)>0, scnt as counter)
 writeln(scnt, ": ", s)
```

And here is an elegant formulation how to calculate the average value of set elements with a given property (odd numbers):

```
S:= {1, 5, 8, -1, 4, 7, 2}
cnt:=0.0
writeln("Average of odd numbers: ",
        (sum(cnt as counter, i in S | isodd(i)) i) / cnt)
```

As an alternative to adding a counter on a loop, Mosel also defines the *aggregate operator* `count` that is used as follows.

```
writeln("Number of odd numbers in S: ", count(i in S | isodd(i)) )

writeln("Occurences of 'b' in L: ", count(s in L | findtext(s, 'b', 1)>0) )
```

Both types of counters may be used jointly in a single statement as shown in the following example (model `count2.mos`) that creates an entry for the array `NODE` if there are at least two incoming or outgoing arcs for the corresponding index `j`.

```
declarations
  I: set of integer
  ARC: dynamic array(I,I) of boolean
  NODE: dynamic array(set of integer) of integer
end-declarations

initializations from "count2.dat"
  ARC
end-initializations

ctnode:=0
forall(ctnode as counter, j in I |
       count(i in I | exists(ARC(i,j))) +
       count(i in I | exists(ARC(j,i))) >= 2) create(NODE(j))

writeln("Number of nodes created: ", ctnode)
```

7.2.2 with

The `with` statement is handled like a single iteration of a `forall` loop. It allows the developer to work with local declarations for a block of code that needs to be surrounded by `do` / `end-do` even if it only contains a single statement. Such local declarations can be used, for example, to simplify and speed up repeated access to specific elements of complex data structures as shown in the code snippet below (file `usingwith.mos`).

```
declarations
  NB: dynamic array(R:range) of set of integer
end-declarations

NB(1):={2,3,4}; NB(2):={1,3,5,6}; NB(6):={2,4}
forall(i in R | exists(NB(i)) )
  with S=NB(i), el=getelt(S) do
    writeln(i, ": Size of set=", S.size, ", contains 1:", 1 in S,
            ", an element of the set:", el, " is different from 0:", el<>0)
  end-do
```

Note that `forall` loop statements equally accept local declarations of scalars (but not of structured

types). In the example above the declaration of the scalar `el` could be moved directly into the `forall` statement, but not the local declaration of the set `S`:

```
forall(i in R | exists(NB(i)), el=getelt(NB(i)))
  with S=NB(i) do
    ...
```

Local declarations in `with` statements can also serve for aliasing arrays along with their index set(s), for example when a program needs to enumerate an array that has been declared without naming its indexing sets (note also the use of `exists` with the locally declared entities in this example):

```
declarations
  A1: dynamic array(range,string) of integer  ! Array with unnamed index sets
end-declarations

A1(1,'a'):=1; A1(4,'c'):=4; A1(6,'d'):=6

with A(I,J)=A1 do
  forall(i in I, j in J | exists(A(i,j))) write(" ", A(i,j))
end-do
writeln
```

7.2.3 while

A `while` loop is typically employed if the number of times that the loop needs to be executed is not know beforehand but depends on the evaluation of some condition: a set of statements is repeated while a condition holds. As with `forall`, the `while` statement exists in two versions, an inline version (`while`) and a version (`while-do`) that is to be used with a block of program statements.

The following example (model `lcdiv1.mos`) computes the largest common divisor of two integer numbers `A` and `B` (that is, the largest number by which both `A` and `B`, can be divided without remainder). Since there is only a single `if-then-else` statement in the `while` loop we could use the inline version of the loop but, for clarity's sake, we have given preference to the `while-do` version that marks where the loop terminates clearly.

```
model Lcdiv1

declarations
  A,B: integer
end-declarations

write("Enter two integer numbers:\n  A: ")
readln(A)
write("  B: ")
readln(B)

while (A <> B) do
  if A>B then
    A:=A-B
  else B:=B-A
  end-if
end-do

writeln("Largest common divisor: ", A)

end-model
```

7.2.4 repeat until

The `repeat-until` structure is similar to the `while` statement with the difference that the actions in the loop are executed once before the termination condition is tested for the first time.

The following example (model `shsort.mos`) combines the three types of loops (`forall`, `while`, `repeat-until`) that are available in Mosel. It implements a *Shell sort* algorithm for sorting an array of numbers into numerical order. The idea of this algorithm is to first sort, by straight insertion, small groups of numbers. Then several small groups are combined and sorted. This step is repeated until the whole list of numbers is sorted.

The spacings between the numbers of groups sorted on each pass through the data are called the increments. A good choice is the sequence which can be generated by the recurrence $inc_1 = 1,\ inc_{k+1} = 3 \cdot inc_k + 1,\ k = 1, 2, \ldots$

```
model "Shell sort"

 declarations
   N: integer                   ! Size of array ANum
   ANum: array(range) of real   ! Unsorted array of numbers
 end-declarations

 N:=50
 forall(i in 1..N) ANum(i):=round(random*100)

 writeln("Given list of numbers (size: ", N, "): ")
 forall(i in 1..N) write(ANum(i), " ")
 writeln

 inc:=1                         ! Determine the starting increment
 repeat
   inc:=3*inc+1
 until (inc>N)

 repeat                         ! Loop over the partial sorts
   inc:=inc div 3
   forall(i in inc+1..N) do     ! Outer loop of straight insertion
     v:=ANum(i)
     j:=i
     while (ANum(j-inc)>v) do   ! Inner loop of straight insertion
       ANum(j):=ANum(j-inc)
       j -= inc
       if j<=inc: break
     end-do
     ANum(j):= v
   end-do
 until (inc<=1)

 writeln("Ordered list: ")
 forall(i in 1..N) write(ANum(i), " ")
 writeln

end-model
```

The example introduces a new statement: `break`. It can be used to interrupt one or several loops. In our case it stops the inner `while` loop. Since we are jumping out of a single loop, we could as well write `break 1`. If we wrote `break 3`, the `break` would make the algorithm jump 3 loop levels higher, that is outside of the `repeat-until` loop.

Note that there is no limit to the number of nested levels of loops and/or selections in Mosel.

CHAPTER 8

Arrays, sets, lists, records, and user types

The Mosel language defines the structured types set, array, list, and record. So far we have worked with arrays and sets relying on an intuitive understanding of what is an 'array' or a 'set'. More formally, we may define an *array* as a collection of labeled objects of a given type where the label of an array entry is defined by its index tuple.

A *set* collects objects of the same type without establishing an order among them (as opposed to arrays and lists). Set elements are unique: if the same element is added twice the set still only contains it once.

A *list* groups objects of the same type. Unlike sets, a list may contain the same element several times. The order of the list elements is specified by construction.

Mosel arrays, sets and lists may be defined for any type, that is the elementary types (including the basic types `integer`, `real`, `string`, `boolean` and the MP types `mpvar` and `linctr`), structured types (`array`, `set`, `list`, `record`), and external types (contributed to the language by a module).

A *record* is a finite collection of objects of any type. Each component of a record is called a *field* and is characterized by its name and its type.

This chapter first presents in a more systematic way the different possibilities of how arrays and sets may be initialized (all of which the reader has already encountered in the examples in the first part), and also shows more advanced ways of working with sets. We then introduce lists, showing how to initialize and access them, and finally give some examples of the use of records.

The last two sections deal with the definition of *user types*, including how to work with *unions*.

8.1 Arrays

In the first part of this manual we have already encountered many examples that make use of arrays. The most important points are summarized in this section and here is an overview of the topics explained with other examples:

- The initialization operator `::` and value assignment: Section 2.1
- Multidimensional arrays: Section 2.1
- String indices: Section 2.1.3
- Initialization from file:
  – dense format text file (Section 2.2),
  – ODBC connection (Section 2.2.5),
  – Excel spreadsheets (Section 2.2.5.1),
  – sparse format text file (Section 3.2),
  – alternative text formats (Section 3.4)
- Dynamic variable creation and finalization: Section 3.3.1

8.1.1 Array declaration

Here are some examples of array definition:

```
declarations
  A1: array(1..3) of integer        ! Fixed size array
  F = {"a","b","c"}
  A2: array(F) of real              ! Fixed size array
  A3: array(R:range) of integer     ! Dense array with unknown index set
  A4: dynamic array(F) of real      ! Dynamic array
end-declarations

writeln("A1:", A1, " A2:", A2, " A3:", A3, " A4:", A4, " A5:", A5)

! Using the array initialization operator
A1::[10,20,30]                      ! Range indices are known
A2::(["a","b","c"])[1.1, 2.5, 3.9]  ! String indices must be stated
A3::(1..3)[10,20,30]                ! Indices are not known upfront

A2("a"):=5.1                        ! Redefine an entry

setrandseed(3)
forall(f in F) A4(f):= 10*random    ! Value assignment
delcell(A4("a"))                    ! Deleting an array entry

writeln("A1:", A1, " A2:", A2, " A3:", A3, " A4:", A4)
```

The output produced by this model (file `arraydef.mos`) is the following.

```
A1:[0,0,0] A2:[0,0,0] A3:[] A4:[]
A1:[10,20,30] A2:[5.1,2.5,3.9] A3:[(1,10),(2,20),(3,30)] A4:[( `b',7.6693),( `c',5.89101)]
```

Arrays A1 and A2 are *fixed size arrays*: their size (*i.e.* the total number of objects/cells they contain) is known at their declaration because all their indexing sets are of fixed size (*i.e.* either constant or finalized). All the cells of fixed size arrays are created and initialized immediately, using default initialization values that depend on the array type. For Mosel's basic types these are the following values.
`real`, `integer`: 0
`boolean`: `false`
`string`: `''` (*i.e.* the empty string)

Array A4 is explicitly marked as *dynamic array* using the qualifier `dynamic`. Dynamic arrays along with hashmap arrays are the two forms of *sparse arrays* in Mosel, a hashmap array is obtained by applying the qualifier `hashmap` in place of `dynamic`—both types are used in the same way, but their performance differs (dynamic arrays are generally faster for linear enumeration and require less memory whereas hashmap arrays are faster for random access). Sparse arrays are created empty. Their cells are created explicitly (see Paragraph 8.1.1.2 below) or when they are assigned a value, that is, the array size will grow 'on demand'. It is also possible to delete some or all cells of a sparse array using the procedure `delcell` on an entry or the whole array (same as `reset`). The value of a cell that has not been created is the default initial value of the type of the array.

Array A3 is created empty since its indexing set is empty at the time of its declaration, but this array is not the same as a dynamic array. It is a dense array that will grow if elements are added to its index set. Please refer to Appendix B.3 for further detail.

8.1.1.1 Multiple indices and array access functions

Arrays with multiple indices are defined and accessed as follows:

```
declarations
 C: array(range, set of string, set of real) of integer
 D: array(1..5) of array(range, set of string) of real
end-declarations

C(5,"ab",1.5):= 10
C(5,"dce",2.5):= 15
writeln("C=", C, ", size of C: ", C.size, ", number of dimensions: ", C.nbdim)
forall(i in 1..getnbdim(C))
 writeln("  index set ", i, " = ", C.index(i), " (size:", C.index(i).size, ")")

! Alternative syntax for accessing entries of 'array of array' structures
D(1,7,"a"):= 2.8
D(1)(9,"b"):= 20
```

As shown in the example, in order to access (or 'dereference') the cell of an *array of arrays* such as the array `D`, the list of indices for the second array can either be appended to the list of indices of the first array, or alternatively, a second set of parantheses can be used for the index tuple of the second array.

Mosel defines various **access functions** for retrieving information about arrays (`getsize`, `getnbdim`, `getindex`) that can also be used in their dot notation form as shown in the example. The code snippet results in the following output:

```
C=[10,0,0,15], size of C: 4, number of dimensions: 3
  index set 1 = 5..5 (size:1)
  index set 2 = { `ab', `dce'} (size:2)
  index set 3 = {1.5,2.5} (size:2)
```

The declaration of the arrays in the code snippet above employs several different **types of index sets**: the most common index set types probably are range sets, and sets of types integer or string. Mosel accepts any type of set as array index (including sets of structured types), however, for most pratical purposes it is recommended to employ only *constant types* as array indices (that is, the four basic types `integer`, `string`, `boolean`, `real`, or external types such as `date`/`time`/`datetime` that support the 'constant' property: see the example in Section 17.5.2). Note that while it is possible to use index sets of type `real` for Mosel arrays this is not a generally encouraged practice: due to the underlying floating point representation it is not always guaranteed that two index values that look the same are indeed identical.

8.1.1.2 `create`

Special care needs to be taken in the case of sparse arrays of decision variables (and indeed with any types that do not have an assignment operator). Writing `x:=1` is a syntax error if `x` is of type `mpvar`. If an array of such a type is defined as dynamic or hashmap array, then the corresponding cells are not created. The entries of the array must be created explicitly by using the procedure `create` since they cannot be defined by assignment. Let us simply recall here the example from Section 3.2.

```
declarations
  REGION: set of string               ! Set of customer regions
  PLANT: set of string                ! Set of plants
  TRANSCAP: dynamic array(PLANT,REGION) of real
                                      ! Capacity on each route plant->region
  flow: dynamic array(PLANT,REGION) of mpvar    ! Flow on each route
end-declarations

initializations from 'transprt.dat'
  TRANSCAP
end-initializations

! Create the flow variables that exist
forall(p in PLANT, r in REGION | exists(TRANSCAP(p,r)) ) create(flow(p,r))
```

For a more detailed discussion of decision variable creation please see Section 3.3.1.

8.1.2 Array initialization from file

When working with arrays, we distinguish between *dense* and *sparse* data formats. Dense data format means that only the data values are represented (see also Section 2.2); in sparse format each data entry is accompanied by its index tuple. Dense format data uses less storage space but this format can only be used if all indices are defined in the model and if no ambiguity results from the omission of the indices when transferring data. In all other cases sparse data format must be used and it is particularly recommended to use this representation if only few entries of a multidimensional array are actually defined.

```
declarations
  A: array(1..2,1..3) of real          ! Can use dense format
  B: array(R:range,T:range) of real    ! Requires sparse format
  D: dynamic array(set of string, range) of real   ! Requires sparse format
  S: set of string
  M: dynamic array(S) of integer       ! Requires sparse format
  N: dynamic array(S) of string        ! Requires sparse format
end-declarations

initializations from "arrayinit.dat"
  A  B
  D as "SomeName"                      ! Data label different from model name
  D as "SomeName2"                     ! Add some more data to 'D'
  [M,N] as "MNData"                    ! 2 arrays read from the same table
end-initializations

writeln("A:", A, " B:", B, "\nD:", D, "\nM:", M, "\nN:", N)
```

With this contents of the data file `arrayinit.dat`

```
A: [2 4 6 8 10 12]
B: [(1 1) 2 (1 2) 4 (1 3) 6 (2 1) 8 (2 2) 10 (2 3) 12]
SomeName: [("a" 1) 2 ("a" 2) 4 ("b" 3) 6 ("c" 4) 8 ("b" 5) 10]
SomeName2: [("a" 3) 12 ("b" 2) 14 ("b" 5) 16]
MNData: [ ("A") [2 "a"] ("B") [* "b"]
          ("C") [6 *]   ("D") [8 "c"]
          ("E") [10 "b"] ]
```

we see the following output display when executing the model `arrayinit.mos` shown above:

```
A:[2,4,6,8,10,12] B:[2,4,6,8,10,12]
D:[( `a',1,2),( `a',2,4),( `a',3,12),( `b',2,14),( `b',3,6),( `b',5,16),( `c',4,8)]
M:[( `A',2),( `C',6),( `D',8),( `E',10)]
N:[( `A',a),( `B',b),( `D',c),( `E',b)]
```

By default, Mosel expects that data labels are the same as the model names. For array `D` we show how to read data using different labels. The contents of the second set of data labeled `SomeName2` is added to what is read from `SomeName`. Note that the entry (b,5) is contained in both sets, and the corresponding array entry takes its value from the last label that is read.

Arrays such as `M` and `N`, that share the same index sets (but not necessarily the same entries) can be read from a single label/data table. The '`*`' in certain entries of `MNData` indicates that the entry does not exist in one of the arrays.

The syntax of `initializations` blocks remains the same when switching to other data sources. Sections 2.2.5 and 2.2.5.1 discuss examples of using databases or spreadsheets instead of text files for array initialization. For further detail on data I/O using different data sources the reader is refered to the Xpress whitepaper *Using ODBC and other database interfaces with Mosel*.

8.1.3 Automatic arrays: the `array` operator

The keyword `array` can be used as an aggregate operator in order to create an array that will exist only for the duration of the expression. This *automatic array* may be used wherever a reference to an array is expected, for instance, in function calls or in initializations blocks.

In the following example we use the `array` operator to extract the (1-dimensional) rows and column arrays from a 2-dimensional array, we further generate a subarray with a selection of entries and the transposed (inversed indices) array.

```
model "Automatic arrays"

 declarations
   B: array(S:set of string, I:set of real) of integer
 end-declarations

 B::(["a","b"], [3,1.5,7])[1,2,3,4,5,6]
 writeln("B: ", B)

 forall(s in S) writeln("Row ", s, ": ", array(i in I) B(s,i))
 forall(i in I) writeln("Column ", i, ": ", array(s in S) B(s,i))

 writeln("B filtered: ", array(s in S,i in I | s<>"a" and i<5) B(s,i))

 writeln("Transpose: ", array(i in I, s in S) B(s,i))
end-model
```

And this is the output generated by the model `autoarray.mos`.

```
B: [1,2,3,4,5,6]
Row a: [1,2,3]
Row b: [4,5,6]
Column 3: [1,4]
Column 1.5: [2,5]
Column 7: [3,6]
B filtered: [( `b',3,4),( `b',1.5,5)]
Transpose: [1,4,2,5,3,6]
```

As it has been mentioned in Section 8.1.1.1 the use index sets of type `real` for Mosel arrays is not a generally encouraged practice: due to the underlying floating point representation it is not always guaranteed that two index values that look the same are indeed identical.

On the topic of output to file using `initializations to`, see Chapter 10, and particularly the note on solution output using arrays generated 'on the fly' in combination with `evaluation of` in Section 10.2.4.

8.2 Initializing sets

In the revised formulation of the burglar problem in Chapter 2 and also in the models in Chapter 3 we have already seen different examples for the use of index sets. We recall here the relevant parts of the respective models.

8.2.1 Constant sets

In the Burglar example the index set is assigned directly in the model:

```
declarations
 ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
        "chest", "brick"}
end-declarations
```

Since in this example the set contents is set in the declarations section, the index set `ITEMS` is a *constant set* (its contents cannot be changed). To declare it as a *dynamic set*, the contents needs to be assigned after its declaration:

```
declarations
  ITEMS: set of string
end-declarations

ITEMS:={"camera", "necklace", "vase", "picture", "tv", "video",
        "chest", "brick"}
```

8.2.2 Set initialization from file, finalized and fixed sets

In Chapter 4 the reader has encountered several examples how the contents of sets may be initialized from data files.

The contents of the set may be read in directly as in the following case:

```
declarations
  WHICH: set of integer
end-declarations

initializations from 'idata.dat'
  WHICH
end-initializations
```

Where `idata.dat` contains data in the following format:

```
WHICH: [1 4 7 11 14]
```

Unless a set is constant (or finalized), arrays that are indexed by this set (and that are not explicitly marked as sparse arrays) are created as non-fixed dense arrays. Since in many cases the contents of a set does not change any more after its initialization, Mosel's *automatic finalization* mechanism finalizes the set `WHICH` in the `initializations from` block. Consider the continuation of the example above:

```
declarations
  x: array(WHICH) of mpvar
end-declarations
```

The array of variables `x` will be created as a static array since its index set is finalized. Declaring arrays in the form of static arrays is preferable if the indexing set is known before because this allows Mosel to handle them in a more efficient way.

Index sets may also be initialized indirectly during the initialization of non-fixed or sparse arrays:

```
declarations
  REGION: set of string
  DEMAND: array(REGION) of real
end-declarations

initializations from 'transprt.dat'
  DEMAND
end-initilizations
```

If file `transprt.dat` contains the data:

```
DEMAND: [(Scotland) 2840 (North) 2800 (West) 2600 (SEast) 2820 (Midlands) 2750]
```

then printing the set `REGION` after the initialization will give the following output:

```
{ `Scotland', `North', `West', `SEast', `Midlands'}
```

Once a set is used for indexing an array (of data, decision variables *etc.*) it is *fixed*, that is, its elements can no longer be removed, but it may still grow in size.

The indirect initialization of (index) sets is not restricted to the case that data is input from file. In the

following example (model `chess2.mos`) we add an array of variable descriptions to the chess problem introduced in Chapter 1. These descriptions may, for instance, be used for generating a nice output. Since the indexing set `Allvars` of array `DescrV` is not known at declaration time the resulting array is not fixed and both grow with each new variable description that is added to `DescrV`.

```
model "Chess 2"
 uses "mmxprs"

 declarations
   Allvars: set of mpvar
   DescrV: array(Allvars) of string
   small, large: mpvar
 end-declarations

 DescrV(small):= "Number of small chess sets"
 DescrV(large):= "Number of large chess sets"

 Profit:=  5*small + 20*large
 Lathe:=   3*small + 2*large <= 160
 Boxwood:=   small + 3*large <= 200

 maximize(Profit)

 writeln("Solution:\n Objective: ", getobjval)
 writeln(DescrV(small), ": ", getsol(small))
 writeln(DescrV(large), ": ", getsol(large))

end-model
```

The reader may have already remarked another feature that is illustrated by this example: the indexing set `Allvars` is of type `mpvar`. So far only basic types have occurred as index set types but as mentioned earlier, sets in Mosel may be of any elementary type, including the MP types `mpvar` and `linctr`.

8.3 Working with sets

In all examples of sets given so far sets are used for indexing other modeling objects. But they may also be used for different purposes.

The following example (model `setops.mos`) demonstrates the use of basic set operations in Mosel: *union* (+), *intersection* (*), and *difference* (–):

```
model "Set example"

 declarations
   Cities={"rome", "bristol", "london", "paris", "liverpool"}
   Ports={"plymouth", "bristol", "glasgow", "london", "calais",
          "liverpool"}
   Capitals={"rome", "london", "paris", "madrid", "berlin"}
 end-declarations

 Places:= Cities+Ports+Capitals        ! Create the union of all 3 sets
 In_all_three:= Cities*Ports*Capitals  ! Create the intersection of all 3 sets
 Cities_not_cap:= Cities-Capitals      ! Create the set of all cities that are
                                       ! not capitals

 writeln("Union of all places: ", Places)
 writeln("Intersection of all three: ", In_all_three)
 writeln("Cities that are not capitals: ", Cities_not_cap)

end-model
```

The output of this example will look as follows:

```
 Union of all places:{ `rome', `bristol', `london', `paris', `liverpool',
  `plymouth', `bristol', `glasgow', `calais', `liverpool', `rome', `paris',
  `madrid', `berlin'}
 Intersection of all three: { `london'}
 Cities that are not capitals: { `bristol', `liverpool}
```

Sets in Mosel are indeed a powerful facility for programming as in the following example (model `prime.mos`) that calculates all *prime numbers* between 2 and some given limit.

Starting with the smallest one, the algorithm takes every element of a set of numbers `SNumbers` (positive numbers between 2 and some upper limit that may be specified when running the model), adds it to the set of prime numbers `SPrime` and removes the number and all its multiples from the set `SNumbers`.

```
model Prime

 parameters
   LIMIT=100                    ! Search for prime numbers in 2..LIMIT
 end-parameters

 declarations
   SNumbers: set of integer     ! Set of numbers to be checked
   SPrime: set of integer       ! Set of prime numbers
 end-declarations

 SNumbers:={2..LIMIT}

 writeln("Prime numbers between 2 and ", LIMIT, ":")

 n:=2
 repeat
   while (not(n in SNumbers)) n+=1
   SPrime += {n}                ! n is a prime number
   i:=n
   while (i<=LIMIT) do          ! Remove n and all its multiples
     SNumbers-= {i}
     i+=n
   end-do
 until SNumbers={}

 writeln(SPrime)
 writeln(" (", getsize(SPrime), " prime numbers.)")

end-model
```

This example uses a new function, `getsize`, that if applied to a set returns the number of elements of the set. The condition in the `while` loop is the logical negation of an expression, marked with `not`: the loop is repeated as long as the condition `n in SNumbers` is not satisfied.

8.3.1 Set operators

The preceding example introduces the operator += to add sets to a set (there is also an operator −= to remove subsets from a set). Another set operator used in the example is `in` denoting that a single object is contained in a set. We have already encountered this operator in the enumeration of indices for the `forall` loop.

Mosel also defines the standard operators for comparing sets: subset (<=), superset (>=), difference (<>), end equality (=). Their use is illustrated by the following example (model `setcomp.mos`):

```
model "Set comparisons"

 declarations
   RAINBOW = {"red", "orange", "yellow", "green", "blue", "purple"}
```

```
  BRIGHT = {"yellow", "orange"}
  DARK = {"blue", "brown", "black"}
end-declarations

writeln("BRIGHT is included in RAINBOW: ", BRIGHT <= RAINBOW)
writeln("RAINBOW is a superset of DARK: ", RAINBOW >= DARK)
writeln("BRIGHT is different from DARK: ", BRIGHT <> DARK)
writeln("BRIGHT is the same as RAINBOW: ", BRIGHT = RAINBOW)

end-model
```

As one might have expected, this example produces the following output:

```
BRIGHT is included in RAINBOW: true
RAINBOW is a superset of DARK: false
BRIGHT is different from DARK: true
BRIGHT is the same as RAINBOW: false
```

8.4 Initializing lists

Lists are not commonly used in the standard formulation of Mathematical Programming problems. However, this data structure may be useful for the Mosel implementation of some more advanced solving and programming tasks.

8.4.1 Constant list

If the contents of a list are specified at the declaration of the list, such as

```
declarations
  L = [1,2,3,4,5,6,7,8,9,10]
end-declarations
```

we have defined a *constant list* (its contents cannot be changed). If we want to be able to modify the list contents subsequently we need to separate the definition of the list contents from the declaration, resulting in a *dynamic list*:

```
declarations
  L: list of integer
end-declarations

L:= [1,2,3,4,5,6,7,8,9,10]
```

A two-dimensional array of lists may be defined thus (and higher dimensional arrays by analogy):

```
declarations
  M: array(range,set of integer) of list of string
end-declarations

M:: (2..4,1)[['A','B','C'], ['D','E'], ['F','G','H','I']]
```

8.4.2 List initialization from file

Similarly to what we have already seen for other data structures, the contents of lists may be initialized from file through `initializations` blocks. For example,

```
declarations
  K: list of integer
  N: array(range,set of integer) of list of string
end-declarations
```

```
initializations from "listinit.dat"
  K  N
end-initializations

writeln("K: ", K)
writeln("An entry of N: ", N(5,3))
```

Assuming the datafile `listinit.dat` contains these lines

```
K: [5 4 3 2 1 1 2 3 4 5]

N: [(3 1) ['B' 'C' 'A']
    (5 3) ['D' 'E']
    (6 1) ['H' 'I' 'F' 'G']]
```

we obtain the following output from the model fragment above:

```
K: [5,4,3,2,1,1,2,3,4,5]
An entry of N: [ `D', `E']
```

8.5 Working with lists

8.5.1 Enumeration

Similarly to the way we have used sets so far, lists may be used as loop indices for enumeration. The following enumerates a given list `L` from beginning to end:

```
declarations
  L: list of integer
end-declarations

L:= [1,2,3,4,5]

forall(i in L) writeln(i)
```

Since lists have an ordering we may choose, for instance, to reverse the order of list elements for the enumeration. The model `listenum.mos` below shows several possibilities for enumerating lists in inverse order: (1) reversing a copy of the list to enumerate, (2) reversing the list to enumerate. In the first case we obtain the reversed copy of the list with function `getreverse`, in the second case we modify the original list by applying to it the procedure `reverse`.

```
model "Reversing lists"

declarations
  K,L: list of integer
end-declarations

L:= [1,2,3,4,5]

! Enumeration in inverse order:
! 1. Reversed copy of the list (i.e., no change to 'L')
K:=getreverse(L)
forall(i in K) writeln(i)

! 2. Reversing the list itself
reverse(L)
forall(i in L) writeln(i)

end-model
```

8.5.2 List operators

Lists are composed by concatenating several lists or by truncating their extremities (refered to as *head* and *tail*). The operators += and + serve for concatenating lists. Their inverses (−= and −) may be used to remove the tail of a list—they will not remove the given sublist if it is not positioned at the end.

The following model `listops.mos` shows some examples of the use of list operators. Besides the concatenation operators + and += we also use the aggregate form `sum`. Another list operator used in this example is the comparison operator <> (the comparison operator = may also be used with lists).

```
model "List operators"

 declarations
   L,M: list of integer
 end-declarations

 L:= [1,2,3] + [4,5]; writeln("L (1): ", L)
 L+= [6,7,8]; writeln("L (2): ", L)
 L-= [1,2,3]; writeln("L (3): ", L)

 M:= sum(l in L) [l*2]; writeln("M: ", M)

 writeln("L and M are different: ", L<>M)

end-model
```

As can be seen in the output, the list `[1,2,3]` is not removed from `L` since it is not located at its tail:

```
L (1): [1,2,3,4,5]
L (2): [1,2,3,4,5,6,7,8]
L (3): [1,2,3,4,5,6,7,8]
M: [2,4,6,8,10,12,14,16]
L and M are different: true
```

8.5.3 List handling functions

The Mosel subroutines for list handling form two groups, namely

- Operations preserving the list they are applied to: retrieving a list element (`getelt`, `getfirst`, `getlast`), occurrence of an element (`findfirst`, `findlast`), retrieving a copy of the head or tail (`gethead`, `gettail`), reversed copy of a list (`getreverse`)
- Operations modifying the list they are applied to: cutting off (=discard) individual elements or the head or tail (`cutelt`, `cutfirst`, `cutlast`, `cuthead`, `cuttail`), splitting off (=retrieve) the head or tail (`splithead`, `splittail`), reverse the list (`reverse`)

The following example `listmerge.mos` merges two lists of integers `K` and `L`, the elements of which are ordered in increasing order of their values into a new list `M` that is ordered in the same way. The elements of the two original lists are added one-by-one to the new list using the concatenation operator +=. Whilst the elements of the list `K` are simply enumerated, we iteratively split off the first element from list `L` (using `splithead` with second argument `1` to take away just the first list element) so that this list will be empty at the end of the `forall` loop. If this is not desired, we need to work with a copy of this list.

```
model "Merging lists"

 declarations
   K,L,M: list of integer
 end-declarations

 K:= [1,4,5,8,9,10,13]
 L:= [-1,0,4,6,7,8,9,9,11,11]
```

```
forall(k in K) do
  while (L<>[] and k >= getfirst(L))  M += splithead(L,1)
  M+= [k]
end-do

writeln(M)

end-model
```

The resulting list `M` is:

```
[-1,0,1,4,4,5,6,7,8,8,9,9,9,10,11,11,13]
```

List handling routines provide a powerful means of programming, illustrated by the following example `euler.mos` that constructs a Eulerian circuit for the network shown in Figure 8.1 (thick arrows indicate that the corresponding arc is to be used twice). This example is an alternative implementation of the Eulerian circuit algorithm described in Section 15.4 'Gritting roads' (problem `j4grit`) of the book 'Applications of optimization with Xpress-MP'.

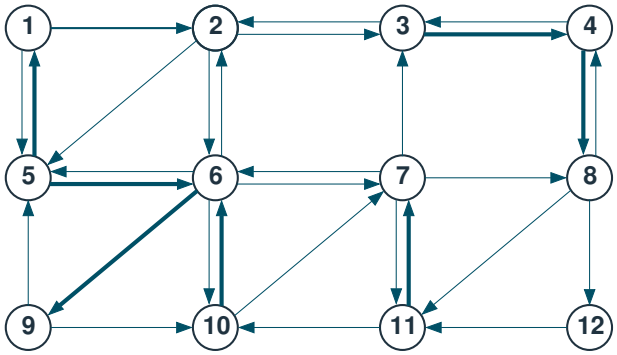


Figure 8.1: Network forming a Eulerian circuit

A Eulerian circuit is a tour through a network that uses every given arc exactly once. To construct such a circuit for a given set of arcs we may employ the following algorithm

- Choose a start node and add it to the tour.
- **while** there are unused arcs:
  - Find the first node in the tour with unused outgoing arcs.
  - Construct a closed subtour starting from this node.
  - Insert the new subtour into the main tour.

```
model "Eulerian circuit"

declarations
  NODES = 1..12                       ! Set of nodes
  UNUSED: array(NODES) of list of integer
  TOUR: list of integer
  NEWT, TAIL: list of integer
end-declarations

initializations from 'euler.dat'
 UNUSED
end-initializations

ct:=sum(i in NODES) getsize(UNUSED(i))

TOUR:=[1]                             ! Choose node 1 as start point
```

```
while(ct>0) do                          ! While there are unused arcs:
 ! Find first node in TOUR with unused outgoing arc(s)
  node:=0
  forall(i in TOUR)
    if UNUSED(i) <> [] then
      node:=i
      break
    end-if

 ! Insertion position (first occurrence of 'node' in TOUR)
  pos:= findfirst(TOUR, node)

 ! Construct a new subtour starting from 'node'
  cur:=node                             ! Start with current node
  NEWT:=[]
  while(UNUSED(cur) <> []) do
    NEWT+=splithead(UNUSED(cur),1)      ! Take first unused arc
    cur:=getlast(NEWT)                  ! End point of arc is new current node
  end-do

 ! Stop if the subtour is not a closed loop (=> no Eulerian circuit)
  if cur<>node then                     ! Compare start and end of subtour
    writeln("Tour cannot be closed")
    exit(1)
  end-if

 ! Add the new subtour to the main journey
  TAIL:=splittail(TOUR, -pos)           ! Split off the tail from main tour
  TOUR += NEWT + TAIL                   ! Attach subtour and tail to main tour
  ct -= getsize(NEWT)
end-do

writeln("Tour: ", TOUR)                 ! Print the result

end-model
```

The data file `euler.dat` corresponding to the graph in Figure 8.1 has the following contents:

```
UNUSED: [(1) [2 5] (2) [3 5 6] (3) [2 4 4] (4) [3 8 8]
         (5) [1 1 6 6] (6) [2 5 7 9 9 10] (7) [3 6 8 11]
         (8) [4 11 12] (9) [5 10] (10) [6 6 7]
         (11) [7 7 10] (12) [11] ]
```

A Eulerian circuit for this data set is the tour

$1 \to 2 \to 6 \to 5 \to 6 \to 7 \to 8 \to 12 \to 11 \to 7 \to 11 \to 10 \to 7 \to 3 \to 4 \to 3 \to 4 \to 8 \to 4 \to 8 \to 11 \to 7 \to 6 \to 9 \to 5 \to 6 \to 9 \to 10 \to 6 \to 10 \to 6 \to 2 \to 3 \to 2 \to 5 \to 1 \to 5 \to 1$

8.6 Records

Records group Mosel objects of different types. They may be used, for instance, to structure the data of a large-scale model by collecting all information relating to the same object.

8.6.1 Defining records

The definition of a record has some similarities with the `declarations` block: it starts with the keyword `record`, followed by a list of field names and types, and the keyword `end-record` marks the end of the definition. The definition of records must be placed in a `declarations` block. The following code extract defines a record with two fields ('name' and 'values').

```
declarations
 R = 1..10
 D: record
  name: string
  values: array(R) of real
 end-record
end-declarations
```

We need to define a name (*e.g.*, 'mydata') for the record if we want to be able to refer to it elsewhere in the model—note that we declare this record as `public` in order to make all its fields public (so in particular, visible in output display), alternatively, individual fields can be declared as public. For example:

```
declarations
  R = 1..10
  mydata = public record
    name: string
    values: array(R) of real
  end-record
  D: mydata
  A: array(range) of mydata
end-declarations
```

The fields of a record are accessed by appending *`.fieldname`* to the record, for instance:

```
D.name:= "D"
forall(i in R) D.values(i):= i
writeln("Values of ", D.name, ": ", D.values)

writeln("An entry of A: ", A(1))
writeln("'name' of an entry of A: ", A(4).name)
writeln("'values' of an entry of A: ", A(3).values)
writeln("First entry of 'values': ", A(3).values(1))
```

Note: if a record field is an array, the index set(s) of the array can be either constant, or be declared outside of the record definition, or remain unnamed. So, these are all valid record definitions:

```
declarations
  R: range
  P: record
    values: array(R) of real
  end-record

  Q: record
    values: array(1..10) of real
  end-record

  T = record
    values: array(range) of real
  end-record
  L: list of T
end-declarations
```

Working with unnamed index sets for a record field means that each entry of the list `L` has its own index set the contents of which is not influenced by any of the other list entries—if a set name was specified for the array `values` in the record type definition for `T` all entries of `L` would be sharing the same index set.

8.6.2 Initialization of records from file

The contents of a record may be assigned fieldwise within a model as shown above or else be read in from file using `initializations`. The data file must contain the data entries for the different record fields in their order of occurrence in the record definition. An array `A` of the record type `mydata` defined

in the previous section is initialized with data from file in the obvious way (model `recorddef.mos`):

```
declarations
  A: dynamic array(T:range) of mydata
end-declarations

initializations from "recorddef.dat"
  A
end-initializations

writeln(A(1))
forall(i in T | exists(A(i))) writeln(A(i).name)
```

If the data file `recorddef.dat` has these contents:

```
A: [(1) ['A1' [(2) 2 (3) 3 (4) 4] ]
    (3) ['A3' [(3) 6 (6) 9] ]
    (4) ['A4' [5 6 7 8] ]
    (7) ['A7']                      ! Define just the first field
    (6) [* [(6) 6] ]                ! Skip the first field
   ]
```

we obtain the following output (the entry with index 6 is defined but has no name, which accounts for the empty line between 'A4' and 'A7'):

```
[name= `A1' values=[0,2,3,4,0,0,0,0,0,0]]
A1
A3
A4

A7
```

An example of the use of records is the encoding of arcs and associated information such as for representing the network in Figure 8.2.

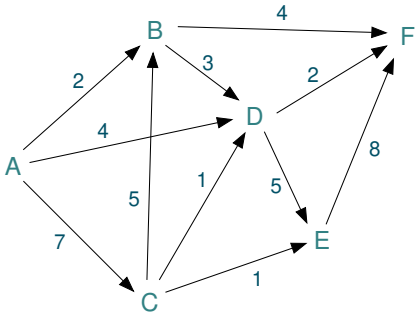


Figure 8.2: Network with costs on arcs

A data file with the network data may look as follows (file `arcs.dat`):

```
ARC: [(1) ["A" "B" 2]
      (2) ["A" "D" 4]
      (3) ["A" "C" 7]
      (4) ["B" "F" 4]
      (5) ["B" "D" 3]
      (6) ["C" "B" 5]
      (7) ["C" "D" 1]
      (8) ["C" "E" 1]
      (9) ["D" "F" 2]
     (10) ["D" "E" 5]
     (11) ["E" "F" 8] ]
```

We may then write our model `arcs.mos` thus

```
model "Arcs"

 declarations
   NODES: set of string                  ! Set of nodes
   ARC: array(ARCSET:range) of record    ! Arcs:
     Source,Sink: string                 !   Source and sink of arc
     Cost: real                          !   Cost coefficient
   end-record
 end-declarations

 initializations from 'arcs.dat'
   ARC
 end-initializations

! Calculate the set of nodes
 NODES:=union(a in ARCSET) {ARC(a).Source, ARC(a).Sink}
 writeln(NODES)

 writeln("Average arc cost: ", sum(a in ARCSET) ARC(a).Cost / getsize(ARCSET) )

end-model
```

The record definition may contain additional fields (*e.g.*, decision variables) that are not to be initialized from file. In this case we need to specify in the `initializations` block which record fields are to be filled with data (example `arcs2.mos`).

```
 declarations
   NODES: set of string                  ! Set of nodes
   ARC: array(ARCSET:range) of record    ! Arcs:
     flow: mpvar                         !   Flow quantity
     Source,Sink: string                 !   Source and sink of arc
     Cost: real                          !   Cost coefficient
   end-record
 end-declarations

 initializations from 'arcs.dat'
   ARC(Source,Sink,Cost)
 end-initializations
```

This functionality can also be used to read separately, and possibly from different sources, the contents of the record fields (example `arcs3.mos`). For instance, the 'Cost' field of our record `ARC` could be initialized as follows:

```
 initializations from 'arcs.dat'
   ARC(Cost) as "COST"
 end-initializations
```

where the data array 'COST' is given as

```
COST: [(1) 2 (2) 4 (3) 7 (4) 4 (5) 3 (6) 5
       (7) 1 (8) 1 (9) 2 (10) 5 (11) 8]
```

8.7 User types

In a Mosel model, the user may define new types that will be treated in the same way as the predefined types of the Mosel language. New types are defined in `declarations` blocks by specifying a type name, followed by =, and the definition of the type. The simplest form of a type definition is to introduce a new name for an existing type, such as:

```
declarations
  myint = integer
  myreal = real
end-declarations
```

In the section on records above we have already seen an example of a user type definition for records (where we have named the record 'mydata'). Another possible use of a user type is as a kind of 'shorthand' where several (data) arrays have the same structure, such as in the model `blend.mos` from Chapter 2, where, instead of

```
declarations
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)
end-declarations
```

we could have written

```
declarations
  ORES = 1..2                    ! Range of ores

  myarray = array(ORES) of real  ! Define a user type

  COST: myarray                  ! Unit cost of ores
  AVAIL: myarray                 ! Availability of ores
  GRADE: myarray                 ! Grade of ores (measured per unit of mass)
end-declarations
```

without making any other modifications to the model.

8.7.1 Type constructors

The following example (file `arcs4.mos`) defines the 'arc' record structure that we have seen in the previous section as a user type and employs this type in the definition of another data structure.

```
public declarations
  NODES: list of string               ! List of nodes
  arc = public record                 ! Arcs:
    Source,Sink: string               !   Source and sink of arc
    Cost: real                        !   Cost coefficient
  end-record
  ARCLIST: list of arc
end-declarations

NODES:=['A','B','C','D','E','F']

i:='S'; ct:=0
forall(j in NODES, ct as counter) do
  ARCLIST+= [arc(.Source:=i,.Sink:=j,.Cost:=10*ct)]
  i:=j
end-do

writeln(ARCLIST)
```

In this example, the type name 'arc' is used as *type constructor* for populating the list `ARCLIST` with 6 different entries, namely:

```
[[Source= `S' Sink= `A' Cost=10],[Source= `A' Sink= `B' Cost=20],
 [Source= `B' Sink= `C' Cost=30],[Source= `C' Sink= `D' Cost=40],
 [Source= `D' Sink= `E' Cost=50],[Source= `E' Sink= `F' Cost=60]]
```

Type constructors for records can be used to populate all or only a selection of the record fields. For example, this code snippet

```
! Initializing selected record fields
 writeln(arc(.Source:="S"))
 writeln(arc(.Sink:="Z"))
 writeln(arc(.Cost:=100,.Source:="XYZ"))
```

will result in the following output:

```
[Source= `S' Sink= `' Cost=0]
[Source= `' Sink= `Z' Cost=0]
[Source= `XYZ' Sink= `' Cost=100]
```

Further examples of type constructors are discussed in Section 17.5 for the creation of date, time, and datetime data.

8.8 Unions

A union is a container capable of holding an object of one of a predefined set of types. One possible use of this functionality is for reading and storing input data of a-priori unknown type. It can also be helpful for advanced programming tasks involving retrieval of information from another Mosel model without prior knowledge of its structure/data model or for inspecting and modifying in a generic way the running model itself (the latter is typically refered to as *reflection*, see the Mosel Language Reference Manual chapter on *mmreflect* for further detail).

8.8.1 Defining unions

Defining an entity of a union type consists in specifying the set of compatible types or the predefined union type `any` than can handle any type without restriction.

```
declarations
  u: string or real               ! Scalar accepting 'string' or 'real'
  a: any                          ! Entity accepting any type
! Defining a type name for the union of the 4 basic Mosel types:
  basictype = string or integer or real or boolean
  B: array(range) of basictype    ! Array of union type 'basictype'
end-declarations
```

The first type that is specified for a union has a special meaning as it defines the representation to use when the union is initialized from some textual form, besides that, order has no impact. In the case of the predefined union type `any` the textual representation is handled by a string. If any of the specified compatible types is itself a union the resulting set of compatible types will include all types of this union.

The *type of a union* is determined when a value is assigned to the entity, it can be retrieved via `gettypeid` or using the operator `is`:

```
declarations
   u: string or real              ! Scalar accepting 'string' or 'real'
 end-declarations
 u:="a"
 writeln(u, " is string: ", u.typeid = string.id)      ! Output:  true
 u:=7.5
 writeln(u, " is real: ", u.typeid = real.id)          ! Output:  true
! The preceding line is equivalent to:
 writeln(u, " is real: ", u is real)                   ! Output:  true
```

It is possible to force a particular type in assignments. You need to employ the `create` procedure for

types that do not support assignment such as 'mpvar':

```
declarations
   a: any                          ! Entity accepting any type
 end-declarations
 a.real:=10
 writeln(a, ":", a is real, a is integer)      ! Output: true false
 a.text:="a text"
 writeln(a, ":", a is text, a is string)       ! Output: true false
 create(a.mpvar)
 writeln(getsol(a.mpvar), ":", a is mpvar)     ! Output: true
```

A union entity can be assigned a compatible union type, but if its type is not compatible with the destination of the assignment a runtime error will be raised. A *wrapping mechanism* is used when a subroutine expects a union but it receives a compatible type or a different union type:

```
declarations
  u: string or real                ! Scalar accepting 'string' or 'real'
  a: any                           ! Entity accepting any type
! Defining a type name for the union of the 4 basic Mosel types:
  basictype = string or integer or real or boolean
  B: array(range) of basictype     ! Array of union type 'basictype'
end-declarations

! This subroutine accepts any union type as its argument
procedure dosomething(aunion: any)
  writeln("In procedure 1: ", aunion)
end-procedure

a:=date(2020,12,24); u:=7.5
dosomething(a)
dosomething(u)

! This subroutine only accepts compatible union types
procedure dosomething2(aunion: basictype)
  writeln("In procedure 2: ", aunion)
end-procedure

a:=2; u:=7.5; B(1):="abc"
dosomething2(a)      ! Type of 'a' now is integer, so it is compatible, but
                     ! this would fail with the previous type 'date' of 'a'
dosomething2(u)      ! Types of 'u' are a subset of union 'basicvalue'
dosomething2(B(1))
```

All our previous examples have shown how to work with scalar types in unions. But the concept of unions and the definition of `any` in particular is not restricted to these: the following code snippet shows how to retrieve information about collection structures (array, set or list) for entities of a union type. With collection structures the `is` operator can be used to check the structure, optionally in combination with the element type. Equivalenty, this information can be accessed via `getstruct` and `geteltype`. The specific access routines for a structure, such as `getnbdim` or `getindex` for arrays, can also be applied.

```
declarations
  L: list of any                                ! List of type 'any'
  a: any                                        ! Entity of type 'any'
  artype = dynamic array(range,string) of real  ! Array type definition
  rectype = public record                       ! Record type definition
    val: real
  end-record
end-declarations

!**** List and set structures
L:=[1.5, "abc", {true,false}, [1,2,3]]
writeln("L=", L)
writeln(L(1), " is real:", L(1) is real)              ! Output: true
writeln(L(3), ": is set:", L(3) is set,               ! Output: true
```

```
  " is set of boolean:", L(3) is set of boolean,          ! Output: true
  " elem. type is boolean:", L(3).eltype = boolean.id)    ! Output: true
writeln(L(4), ": is list:", L(4).struct=STRUCT_LIST,      ! Output: true
  " is list of integer:", L(4) is list of integer,        ! Output: true
  " elem. type is integer:", L(4).eltype = integer.id)    ! Output: true

!**** Defining and accessing arrays
! Optional: explicit type specification (req. if type not used in assignment)
! create(a.artype); a.array(1,"a").real:= 1.5
a.artype(1,"b"):= 1.5; a.artype(3,"b"):= -0.5
a.array(1,"cde").real:= 7.25;       ! Type has been set by prev. assignments
writeln(a, " is array:", a is array,
  " is array of real:", a is array of real)               ! Output: true true
writeln(" is of artype:", a is artype,
  " elem. type is real:", a.eltype = real.id)             ! Output: true true
writeln("  dimensions:", a.array.nbdim, " size:", a.array.size)
                                                          ! Output: dimensions:2 size:3
forall(i in 1..a.array.nbdim)
  writeln("  index", i, "=", a.array.index(i))  ! Output: index1=1..3
                                                !         index2={'b','cde'}
with A(I,J)=a.artype do
  forall(i in I,j in J | exists(A(i,j))) writeln("A(",i,",",j,")=",A(i,j))
end-do
```

And similarly, a union can be of a record type.

```
declarations
  a: any                                          ! Entity of type 'any'
  rectype = public record                         ! Record type definition
    val: real
  end-record
end-declarations

a.rectype.val:=1.25
writeln(a, " is record:", a is record, " is rectype:", a is rectype)
                                                  ! Output: true true
```

8.8.2 Initialization of unions from file

When initializing unions from text format data files only scalar values of basic types are considered. More precisely, integers, reals, and Booleans are assigned to the union; textual values are used to initialize the entity as if it was of the first compatible type of the union (for the union `any` this is a string). An I/O error will be raised if this type does not support initialization.

```
declarations
  L: list of any                  ! List of type 'any'
  mytype = text or integer or real or boolean
  B: array(R:range) of mytype     ! Array of union type 'mytype'
  TNAME: array(integer) of text   ! Type names for display
end-declarations

initializations from "uniondata.dat"
  L  B
end-initializations

TNAME(integer.id):="integer"; TNAME(real.id):="real"
TNAME(string.id):="string"; TNAME(text.id):="text"
TNAME(boolean.id):="boolean"

writeln("L=", L)
forall(i in L) writeln("  ", i, ":", i.typeid, " ", TNAME(i.typeid))
writeln("B=", B)
forall(i in R)
  writeln("  B", i, "=", B(i),":", B(i).typeid, " ", TNAME(B(i).typeid))
```

Assuming the data file 'uniondata.dat' has this contents:

```
L: [10 "abc" true 7.5]
B: [(1) "abc" (3) 5 (4) 5.6 ]
```

then the code snippet above will produce the following output:

```
L=[10,abc,true,7.5]
  10:1 integer
  abc:3 string
  true:4 boolean
  7.5:2 real
B=[abc,?,5,5.6]
  B1=abc:11 text
  B2=?:-1
  B3=5:1 integer
  B4=5.6:2 real
```

When exporting unions, any non-scalar value or types that do not support conversion to string will result in a NIL value ('?' in text format files) in the generated file. Appending the following lines to the previous example

```
declarations
  x: mpvar
end-declarations
L+= [any(x)]

initializations to "unionout.dat"
  L  B
end-initializations
```

will result in an output file 'unionout.dat' with this contents:

```
L: [10 abc true 7.5 ?]
B: [(1) abc ? 5 5.6]
```

8.8.3 Working with the 'reference to' operator

The 'reference to' operator (in symbols: –>) can be defined on any globally declared entities of basic type or reference type, it can be thought of as an *alias* to the referenced entity.

```
declarations
  A: array(range) of integer
end-declarations
A::(2..5)[2,4,6,8]

u:=A                          ! u holds a copy of A
writeln("A=", A, "  u=", u)     ! A=[2,4,6,8]  u=[2,4,6,8]
writeln("u is array:", u is array)   ! Output: true
u.array(2).integer:=10          ! Modify 'u'
writeln("A=", A, "  u=", u)     ! A=[2,4,6,8]  u=[10,4,6,8]   ('A' unchanged)
A(5):=10                        ! Modify 'A'
writeln("A=", A, "  u=", u)     ! A=[2,4,6,10]  u=[10,4,6,8]  ('u' unchanged)

u:=->A                          ! u is a reference to A
writeln("A=", A, "  u=", u)     ! A=[2,4,6,10]  u=[2,4,6,10]
writeln("u is array:", u is array)   ! Output: true
u.array(2).integer:=-1          ! Modify 'u'
writeln("A=", A, "  u=", u)     ! A=[-1,4,6,10]  u=[-1,4,6,10]
A(5):=25                        ! Modify 'A'
writeln("A=", A, "  u=", u)     ! A=[-1,4,6,25]  u=[-1,4,6,25]
! Reset (=delete) the reference , original array A remains unchanged
reset(u)
writeln("u is defined:", isdefined(u), " is array:", u is array)  ! Output:  false  false
```

A special case (and probably the most likely use) are references to subroutines that are discussed in a separate section: see section 9.6 below.

CHAPTER 9

Functions and procedures

When programs grow larger than the small examples presented so far, it becomes necessary to introduce some structure that makes them easier to read and to maintain. Usually, this is done by dividing the tasks that have to be executed into subtasks which may again be subdivided, and indicating the order in which these subtasks have to be executed and which are their activation conditions. To facilitate this structured approach, Mosel provides the concept of *subroutines*. Using subroutines, longer and more complex programs can be broken down into smaller subtasks that are easier to understand and to work with. Subroutines may be employed in the form of procedures or functions. *Procedures* are called as a program statement, they have no return value, *functions* must be called in an expression that uses their return value.

Mosel provides a set of predefined subroutines (for a comprehensive documentation the reader is referred to the Mosel Reference Manual), and it is possible to define new functions and procedures according to the needs of a specific program. A procedure that has occured repeatedly in this document is writeln. Typical examples of functions are mathematical functions like `abs`, `floor`, `ln`, `sin` *etc.*

9.1 Subroutine definition

User defined subroutines in Mosel have to be marked with `procedure` / `end-procedure` and `function` / `end-function` respectively. The return value of a function has to be assigned to `returned` as shown in the following example (model `subrout.mos`).

```
model "Simple subroutines"

 declarations
   a:integer
 end-declarations

 function three:integer
   returned := 3
 end-function

 procedure printstart
   writeln("The program starts here.")
 end-procedure

 printstart
 a:=three
 writeln("a = ", a)

end-model
```

This program will produce the following output:

```
The program starts here.
a = 3
```

9.2 Parameters

In many cases, the actions to be performed by a procedure or the return value expected from a function depend on the current value of one or several objects in the calling program. It is therefore possible to pass parameters into a subroutine. The (list of) parameter(s) is added in parantheses behind the name of the subroutine:

```
function timestwo(b:integer):integer
  returned := 2*b
end-function
```

The structure of subroutines being very similar to the one of `model`, they may also include `declarations` sections for declaring *local parameters* that are only valid in the corresponding subroutine. It should be noted that such local parameters may *mask* global parameters within the scope of a subroutine, but they have no effect on the definition of the global parameter outside of the subroutine as is shown below in the extension of the example 'Simple subroutines'. As in other programming languages, it is not possible to redefine function/procedure parameters in the corresponding subroutine (the declaration of local parameters must not *hide* these parameters). Mosel considers this as a mistake and prints an error message during compilation.

```
model "Simple subroutines"

declarations
  a:integer
end-declarations

function three:integer
  returned := 3
end-function

function timestwo(b:integer):integer
  returned := 2*b
end-function

procedure printstart
  writeln("The program starts here.")
end-procedure

procedure hide_a_1
  declarations
    a: integer
  end-declarations

  a:=7
  writeln("Procedure hide_a_1: a = ", a)
end-procedure

procedure hide_a_2(a:integer)
  writeln("Procedure hide_a_2: a = ", a)
end-procedure

procedure hide_a_3(a:integer)
  declarations
    a: integer
  end-declarations

  a := 15
  writeln("Procedure hide_a_3: a = ", a)
end-procedure

printstart
a:=three
writeln("a = ", a)
a:=timestwo(a)
```

```
 writeln("a = ", a)
 hide_a_1
 writeln("a = ", a)
 hide_a_2(-10)
 writeln("a = ", a)
 hide_a_3(a)
 writeln("a = ", a)

end-model
```

During the compilation we get the error

```
Mosel: E-165 at (34,4) of  `subrout.mos': Declaration of  `a' hides a parameter.
```

This is due to the redefinition of `a` that is passed as an argument into procedure `hide_a_3` and also appears in the `declarations` of this subroutine. We need to modify the definition of this procedure to correct this error, for example by renaming the subroutine argument:

```
 procedure hide_a_3(aa:integer)
```

The program then results in the following output:

```
The program starts here.
a = 3
a = 6
Procedure hide_a_1: a = 7
a = 6
Procedure hide_a_2: a = -10
a = 6
Procedure hide_a_3: a = 15
a = 6
```

9.2.1 Variable number of arguments

A subroutine can be defined to take a *variable number of arguments* by stating a single argument name followed by the symbol `...` as shown in the following code snippet (model `subrout.mos`).

```
! Subroutines with variable number of arguments
 function sumall(Values:...): integer
   returned:= sum(i in Values) i.integer
 end-function

 procedure showint(Optargs:...)
   forall(i as counter, v in Optargs)
     if v is integer: writeln("arg ", i, ": ", v.integer)
 end-procedure

 writeln("sum = ", sumall(1, 2, 3, 4, 5))

 showint(1.5,0,"abc",true,5)
```

which results in the following output:

```
sum = 15
arg 2: 0
arg 5: 5
```

The unique subroutine argument, `Values` or `Optargs` respectively in the example above, is of the type `list of any` (see Section 8.8), when accessing the elements of this list we therefore need to state their expected type in order to be able to perform any operations on them.

9.3 Recursion

The following example (model `lcdiv2.mos`) returns the largest common divisor of two numbers, just like the example 'Lcdiv1' in the previous chapter. This time we implement this task using recursive function calls, that is, from within function `lcdiv` we call again function `lcdiv`.

```
model Lcdiv2

 function lcdiv(a,b:integer):integer
   if a=b then
     returned:=a
   elif a>b then
     returned:=lcdiv(b,a-b)
   else
     returned:=lcdiv(a,b-a)
   end-if
 end-function

 declarations
   A,B: integer
 end-declarations

 write("Enter two integer numbers:\n  A: ")
 readln(A)
 write("  B: ")
 readln(B)

 writeln("Largest common divisor: ", lcdiv(A,B))

end-model
```

This example uses a simple recursion (a subroutine calling itself). In Mosel, it is also possible to use *cross-recursion*, that is, subroutine A calls subroutine B which again calls A. The only pre-requisite is that any subroutine that is called prior to its definition must be declared before it is called by using the `forward` statement (see Section 9.4 below).

9.4 forward

A subroutine has to be 'known' at the place where it is called in a program. In the preceding examples we have defined all subroutines at the start of the programs but this may not always be feasible or desirable. Mosel therefore enables the user to declare a subroutine separately from its definition by using the keyword `forward`. Equivalently to using this keyword, the subroutine can be declared in a `declarations` block. The *declaration* of of a subroutine states its name, the parameters (type and name) and, in the case of a function, the type of the return value. The *definition* that must follow later in the program contains the body of the subroutine, that is, the actions to be executed by the subroutine.

The following example (model `qsort1.mos`) implements a *quick sort* algorithm for sorting a randomly generated array of numbers into ascending order—please note that the implementation discussed here is merely provided as a programming example, we would generally recommend that you use the `qsort` routine of the Mosel module *mmsystem* in your Mosel programs. The procedure `qsort` that starts the sorting algorithm is defined at the very end of the program, it therefore needs to be declared at the beginning, before it is called. Procedure `startqsort` calls the main sorting routine, `qsort`. Since the definition of this procedure precedes the place where it is called there is no need to declare it (but it still could be done). Procedure qsort calls yet again another subroutine, swap.

The idea of the quick sort algorithm is to partition the array that is to be sorted into two parts. The 'left' part containing all values smaller than the partitioning value and the 'right' part all the values that are larger than this value. The partitioning is then applied to the two subarrays, and so on, until all values are sorted.

```
model "Quick sort 1"

 parameters
   LIM=50
 end-parameters

 forward procedure startqsort(L:array(range) of integer)
(! Equivalent form of declaration:
 declarations
   procedure startqsort(L:array(range) of integer)
 end-declarations
!)

 declarations
   T:array(1..LIM) of integer
 end-declarations

 forall(i in 1..LIM) T(i):=round(.5+random*LIM)
 writeln(T)
 startqsort(T)
 writeln(T)

! Swap the positions of two numbers in an array
 procedure swap(L:array(range) of integer,i,j:integer)
   k:=L(i)
   L(i):=L(j)
   L(j):=k
 end-procedure

! Main sorting routine
 procedure qsort(L:array(range) of integer,s,e:integer)
   v:=L((s+e) div 2)              ! Determine the partitioning value
   i:=s; j:=e
   repeat                         ! Partition into two subarrays
     while(L(i)<v) i+=1
     while(L(j)>v) j-=1
     if i<j  then
       swap(L,i,j)
       i+=1; j-=1
     end-if
   until i>=j
                                  ! Recursively sort the two subarrays
   if j<e and s<j: qsort(L,s,j)
   if i>s and i<e: qsort(L,i,e)
 end-procedure

! Start of the sorting process
 procedure startqsort(L:array(r:range) of integer)
   qsort(L,getfirst(r),getlast(r))
 end-procedure

end-model
```

The quick sort example above demonstrates typical uses of subroutines, namely grouping actions that are executed repeatedly (`qsort`) and isolating subtasks (`swap`) in order to structure a program and increase its readability.

The calls to the procedures in this example are nested (procedure `swap` is called from `qsort` which is called from `startqsort`): in Mosel there is no limit as to the number of nested calls to subroutines (it is not possible, though, to define subroutines within a subroutine).

9.5 Overloading of subroutines

In Mosel, it is possible to re-use the names of subroutines, provided that every version has a different number and/or types of parameters. This functionality is commonly referred to as *overloading*.

An example of an overloaded function in Mosel is `getsol`: if a variable is passed as a parameter it returns its solution value, if the parameter is a constraint the function returns the evaluation of the corresponding linear expression using the current solution.

Function `abs` (for obtaining the absolute value of a number) has different return types depending on the type of the input parameter: if an integer is input it returns an integer value, if it is called with a real value as input parameter it returns a real.

Function `getcoeff` is an example of a function that takes different numbers of parameters: if called with a single parameter (of type `linctr`) it returns the constant term of the input constraint, if a constraint and a variable are passed as parameters it returns the coefficient of the variable in the given constraint.

The user may define (additional) overloaded versions of any subroutines defined by Mosel as well as for his own functions and procedures. Note that it is not possible to overload a function with a procedure and *vice versa*.

Using the possibility to overload subroutines, we may rewrite the preceding example 'Quick sort' as follows (model `qsort2.mos`).

```
model "Quick sort 2"

 parameters
   LIM=50
 end-parameters

 forward procedure qsort(L:array(range) of integer)
(! Equivalent form of declaration:
 declarations
   procedure qsort(L:array(range) of integer)
 end-declarations
!)

 declarations
   T:array(1..LIM) of integer
 end-declarations

 forall(i in 1..LIM) T(i):=round(.5+random*LIM)
 writeln(T)
 qsort(T)
 writeln(T)

 procedure swap(L:array(range) of integer,i,j:integer)
  (...)                         (same procedure body as in the preceding example)
 end-procedure

 procedure qsort(L:array(range) of integer,s,e:integer)
  (...)                         (same procedure body as in the preceding example)
 end-procedure

! Start of the sorting process
 procedure qsort(L:array(r:range) of integer)
   qsort(L,getfirst(r),getlast(r))
 end-procedure

end-model
```

The procedure `startqsort` is now also called `qsort`. The procedure bearing this name in the first implementation keeps its name too; it has got two additional parameters which suffice to ensure that the right version of the procedure is called. To the contrary, it is not possible to give procedure `swap` the same name `qsort` because it takes exactly the same parameters as the original procedure `qsort` and hence it would not be possible to differentiate between these two procedures any more.

9.6 Subroutine references

A subroutine reference is an entity that is able to store a reference to a procedure or a function. A subroutine reference is considered as undefined until it has been associated to an actual subroutine via an assignment; the value assigned to the entity must have a compatible type and will usually be the result of the 'reference to' operator –> to prevent calling of the operand. This association may be cancelled using `reset`. A subroutine reference can be used wherever the corresponding type is expected, in particular for an assignment that initializes another subroutine reference, or for specifying a callback routine (see the example in Section 11.1.6).

The following example declares `myfct` as a reference to a function returning a real and expecting a real as parameter; this symbol is then used to hold and invoke the function `div2`. When a subroutine reference is assigned to a generic union entity the type needs to be indicated when calling the subroutine:

```
function div2(r:real):real
  returned:=r/2
end-function

declarations
  myfct: function(real):real        ! Subroutine reference
  u: any
  realfct=function(real):real       ! Subroutine type definition
end-declarations

myfct:=->div2
writeln("div2(10)=", myfct(10))     ! Output:  div2(10)=5

u:=->div2
! In order to call 'u' as a subroutine we need to indicate its type
writeln("div2(10)=", u.realfct(10))
```

Subroutine types can be used like any other types in the declaration of structured Mosel entities:

```
declarations
  L: list of function(real):real               ! List of subroutines
! or equivalently:
!  L: list of realfct
  arfct:array(string) of function(real):real   ! Array of subroutines
end-declarations

L:=[->cos,->sin,->arctan,->abs,->exp,->div2]
forall(i in [-1.0,0.5]) do
  write("i=", i)
  forall(f in L) write(f(i), " ")
  writeln
end-do

! Syntax for calling an element of an array of subroutines with arguments
arfct("cos"):= ->cos
writeln("cos(1)=", arfct("cos")(1))
```

Note that the Mosel compiler does not guarantee which routine is returned if the name of a subroutine refers to several implementations (*i.e.* the routine has been overloaded) and the context does not make it possible to select a particular version. In the example above, the `abs` routine has several versions (it is defined for integer and for real numbers), however, given that the list `L` is declared with the type 'function of real returning a real' the referene to the real-valued version of `abs` is added to the list.

Subroutines can also occur as return type of a function as is shown in the following code snippet (notice the use of `isdefined` to test whether the subroutine has returned a valid subroutine reference).

```
function choose(name:string):function(real):real
  case name of
    "cos": returned:= ->cos
    "abs": returned:= ->abs
    "div": returned:= ->div2
    else writeln("Unknown selection '", name, "'")
  end-case
end-function

fsel:=choose("cos")
if isdefined(->fsel) then
  writeln("cos(1)=", fsel(1))
end-if

fsel:=choose("error")
if not isdefined(->fsel) then
  writeln("no function selected")
end-if
```

Executing this code will produce the following output:

```
cos(1)=0.5403023059
Unknown selection 'error'
no function selected
```

CHAPTER 10

Output

10.1 Producing formatted output

In some of the previous examples the procedures `write` and `writeln` have been used for displaying data, solution values and some accompanying text. To produce better formatted output, these procedures can be combined with the formatting procedure `strfmt`. In its simplest form, `strfmt`'s second argument indicates the (minimum) space reserved for writing the first argument and its placement within this space (negative values mean left justified printing, positive right justified). When writing a `real`, a third argument may be used to specify the maximum number of digits after the decimal point.

For example, if file `fo.mos` contains

```
model FO
 parameters
   r = 1.0         ! A real
   i = 0           ! An integer
 end-parameters

 writeln("i is ", i)
 writeln("i is ", strfmt(i,6) )
 writeln("i is ", strfmt(i,-6) )
 writeln("r is ", r)
 writeln("r is ", strfmt(r,6) )
 writeln("r is ",strfmt(r,10,4) )
end-model
```

and we run Mosel thus:

```
mosel exec fo 'i=123, r=1.234567'
```

we get output

```
i is 123
i is    123
i is 123
r is 1.23457
r is 1.23457
r is     1.2346
```

The following example (model `transport2.mos`) prints out the solution of model 'Transport' (Section 3.2) in table format. The reader may be reminded that the objective of this problem is to compute the product flows from a set of plants (`PLANT`) to a set of sales regions (`REGION`) so as to minimize the total cost. The solution needs to comply with the capacity limits of the plants (`PLANTCAP`) and satisfy the demand `DEMAND` of all regions.

```
 procedure print_table
  declarations
    rsum: array(REGION) of integer    ! Auxiliary data table for printing
    psum,ct,iflow: integer            ! Counters
```

```
  end-declarations

         ! Print heading and the first line of the table
  writeln("\nProduct Distribution\n", "="*20)
  writeln(strfmt("Sales Region",48))
  write(strfmt("",15), "| ")
  forall(r in REGION) write(strfmt(r,9))
  writeln(" |", strfmt("TOTAL",6), " Capacity")
  writeln("-"*80)

        ! Print the solution values of the flow variables and
        ! calculate totals per region and per plant
  ct:=0
  forall(p in PLANT, ct as counter) do
    if ct=2 then
      write(" Plant ", strfmt(p,-8), "|")
    else
      write("       ", strfmt(p,-8), "|")
    end-if
    psum:=0
    forall(r in REGION) do
      iflow:=round(getsol(flow(p,r)))
      psum += iflow
      rsum(r) += iflow
      if iflow<>0 then
        write(strfmt(iflow,9))
      else
        write("      -- ")
      end-if
    end-do
    writeln("  |", strfmt(psum,6), strfmt(round(PLANTCAP(p)),8))
  end-do

        ! Print the column totals
  writeln("-"*80)
  write(strfmt(" TOTAL",-15), "|")
  forall(r in REGION) write(strfmt(rsum(r),9))
  writeln("  |", strfmt(sum(r in REGION) rsum(r),6)
          strfmt(sum(p in PLANT) PLANTCAP(p),8))

        ! Print demand of every region
  write(strfmt(" Demand",-15), "|")
  forall(r in REGION) write(strfmt(round(DEMAND(r)),9))

        ! Print objective function value
  writeln("\n\nTotal cost of distribution = ", strfmt(getobjval/1e6,0,3),
          " million.")

end-procedure
```

Notice the shorthand `"-"*80` meaning that the string '-' is repeated 80 times. This functionality is provided by the module *mmsystem*, however, it is generally more efficient to work with the type `text` for such string operations (see Section 17.6). In particular, it is usually preferrable to work with `textfmt` in place of `strfmt` for producing formatted output of numbers, and `write` statements with several formatted elements can use a single format string via `formattext` as shown in the following extract from the previous model example.

```
        ! Print the column totals
  writeln("-"*80)
  write(formattext("%-15s|"," TOTAL"))
  forall(r in REGION) write(textfmt(rsum(r),9))
  writeln(formattext("  |%6g%8g", sum(r in REGION) rsum(r),
          sum(p in PLANT) PLANTCAP(p)))
```

With the data from Chapter 3 the procedure `print_table` produces the following output:

```
Product Distribution
====================
                                    Sales Region
               |  Scotland    North    SWest    SEast Midlands | TOTAL Capacity
-------------------------------------------------------------------------------
       Corby   |      --        80       --       920     2000  |  3000    3000
 Plant Deeside |      --      1450     1000        --      250  |  2700    2700
       Glasgow |     2840     1270       --        --       --  |  4110    4500
       Oxford  |      --       --     1600      1900      500  |  4000    4000
-------------------------------------------------------------------------------
 TOTAL         |     2840     2800     2600      2820     2750  | 13810   14200
 Demand        |     2840     2800     2600      2820     2750

Total cost of distribution = 81.018 million.
```

10.2 File output

If we do not want the output of procedure `print_tab` in the previous section to be displayed on screen but to be saved in the file `out.txt`, we simply open the file for writing at the beginning of the procedure by adding

```
fopen("out.txt",F_OUTPUT)
```

before the first `writeln` statement, and close it at the end of the procedure, after the last `writeln` statement with

```
fclose(F_OUTPUT)
```

If we do not want any existing contents of the file `out.txt` to be deleted, so that the result table is appended to the end of the file, we need to write the following for opening the file (closing it the same way as before):

```
  fopen("out.txt",F_OUTPUT+F_APPEND)
```

As with input of data from file, there are several ways of outputting data to a file in Mosel. The following example demonstrates three different ways of writing the contents of an array `A` to a file. The last section (10.2.4) shows how to proceed if the data is not readily available in the form of an array but results from the evaluation of an expression (*e.g.*, solution values, function calls).

10.2.1 Data output with `initializations to`

The first method uses the `initializations` block for creating or updating a file in Mosel's `initializations` format.

```
model "Trio output (1)"
 declarations
   A: array(-1..1,5..7) of real
 end-declarations

 A :: [ 2,  4,  6,
       12, 14, 16,
       22, 24, 26]

! First method: use an initializations block
 initializations to "out_1.dat"
   A as "MYOUT"
 end-initializations
end-model
```

File `out_1.dat` will contain the following:

```
'MYOUT': [(-1 5) 2 4 6 (0 5) 12 14 16 (1 5) 22 24 26]
```

If this file contains already a data entry `MYOUT`, it is replaced with this output without modifying or deleting any other contents of this file. Otherwise, the output is appended at the end of it.

Note: For solution output with `initializations to` please see Section 10.2.4 below.

10.2.2 Data output with `writeln`

As mentioned above, we may create freely formatted output files by redirecting the output of `write` and `writeln` statements to a file:

```
model "Trio output (2)"
 declarations
   A: array(-1..1,5..7) of real
 end-declarations

 A :: [ 2,  4,  6,
       12, 14, 16,
       22, 24, 26]

! Second method: use the built-in writeln function
 fopen("out_2.dat", F_OUTPUT)
 forall(i in -1..1, j in 5..7)
   writeln('A_out(', i, ' and ', j, ') = ', A(i,j))
 fclose(F_OUTPUT)
end-model
```

The nicely formatted output to `out_2.dat` results in the following:

```
A_out(-1 and 5) = 2
A_out(-1 and 6) = 4
A_out(-1 and 7) = 6
A_out(0 and 5) = 12
A_out(0 and 6) = 14
A_out(0 and 7) = 16
A_out(1 and 5) = 22
A_out(1 and 6) = 24
A_out(1 and 7) = 26
```

10.2.3 Data output with `diskdata`

As a third possibility, one may use the `diskdata` subroutine from module `mmetc` to write out comma separated value (CSV) files.

```
model "Trio output (3)"
 uses "mmetc"

 declarations
   A: array(-1..1,5..7) of real
 end-declarations

 A :: [ 2,  4,  6,
       12, 14, 16,
       22, 24, 26]

! Third method: use diskdata
 diskdata(ETC_OUT+ETC_SPARSE,"out_3.dat", A)
end-model
```

The output with `diskdata` simply prints the contents of the array to `out_3.dat`, with option `ETC_SPARSE` each entry is preceded by the corresponding indices:

```
-1,5,2
-1,6,4
-1,7,6
0,5,12
0,6,14
```

```
0,7,16
1,5,22
1,6,24
1,7,26
```

Without option `ETC_SPARSE` `out_3.dat` looks as follows:

```
2,4,6
12,14,16
22,24,26
```

Instead of using the `diskdata` subroutine, we may equally use the `diskdata` I/O driver that is defined by the same module, `mmetc`. In the example above we replace the `diskdata` statement by the following `initializations to` block.

```
[ initializations to 'mmetc.diskdata:'
   A as 'sparse,out_3.dat'
 end-initializations
```

10.2.4 Solution output with `initializations to`

In the previous examples we have seen how to write out to a file the contents of a data array defined in Mosel. The free format output with `write`/`writeln` can be applied to any type of expression. However, if we wish to use the `initializations to` functionality for writing out, for instance, the solution values after an optimization run or the result of a Mosel function we need to proceed in a slightly different way from what we have seen so far.

There are two options:

1. Save the solution values or results into a new Mosel object and work with this copy for the file output. For example,

```
declarations
  x: mpvar
  x_sol: real
  y: array(R) of mpvar
  y_sol: array(R) of real
end-declarations
...                           ! Define and solve an optimization problem
x_sol:= x.sol                 ! Retrieve the solution values
forall(i in R) y_sol(i):= y(i).sol
initializations to "out.txt"
  x_sol
  y_sol
end-initializations
```

2. Use the keyword `evaluation` in the `initializations` block.

```
declarations
  x: mpvar
  y: array(R) of mpvar
end-declarations
...                           ! Define and solve an optimization problem
initializations to "out.txt"
  evaluation of x.sol as "x_sol"
  evaluation of array(i in R) y(i).sol as "y_sol"
end-initializations
```

The `array` construct is used in the model extract above to generate a new array 'on the fly'. Its use is similar to aggregate operators such as `sum` or `union`.

The use of the marker `evaluation of` is not restricted to solution values; it allows you to work with any type of expression directly in the initializations block, including results of Mosel functions or computations as shown in the following example `initeval.mos`. This model writes out detailed results for our introductory Chess example (see Section 1.3).

```
model "Evaluations"
 uses "mmxprs"

 declarations
   small,large: mpvar               ! Decision variables: produced quantities
 end-declarations

 Profit:=  5*small + 20*large       ! Objective function
 Lathe:=   3*small + 2*large <= 160  ! Lathe-hours
 Boxwood:=   small + 3*large <= 200  ! kg of boxwood
 small is_integer; large is_integer  ! Integrality constraints

 maximize(Profit)                   ! Solve the problem

 initializations to "chessout.txt"
   evaluation of getparam("XPRS_mipstatus") as "Status"
   evaluation of getobjval as "Objective"
   evaluation of small.sol as "small_sol"
   evaluation of getsol(large) as "large_sol"
   evaluation of Lathe.slack as "Spare time"
   evaluation of Boxwood.act as "Used wood"
   evaluation of Boxwood.act-200 as "Spare wood"
   evaluation of [ small.sol, large.sol ] as "x_sol"
 end-initializations

end-model
```

The resulting output file `chessout.txt` has the following contents:

```
'Status': 6
'Objective': 1330
'small_sol': 2
'large_sol': 66
'Spare time': 22
'Used wood': 200
'Spare wood': 0
'x_sol': [2 66]
```

10.3 Real number format

Whenever output is printed (including matrix export to a file) Mosel uses a representation of floating point numbers with up to 10 digits (C format %.10g). This format may apply rounding when printing large numbers or numbers with many decimals and it is therefore not conservative for repeated reading and writing of the same value. Besides fixed format printing that is discussed in Section 10.1 Mosel defines several output formats for real numbers that generate conservative textual representations of floating point numbers that can be used, for example, to see the exact results of an optimization run or to produce a matrix output file with greater accuracy. Consider the following example (model `numformat.mos`):

```
model "Formatting numbers"
 parameters
   a = 12345678000.0
   b = 12345678048.9
   c = 12.000000045
   d = 12.0
 end-parameters

 setparam("TXTZTOL", false)    ! Disable use of 'zerotol' setting for display
```

```
 writeln("Default: ", a, "  ", b, "  ", c, "  ", d)

 setparam("REALFMT", "%1.2f")       ! Fixed number of decimals
 writeln("%1.2f:  ", a, "  ", b, "  ", c, "  ", d)

! Exact (conservative) representations of floating point numbers
 setparam("REALFMT", "%.17g")
 writeln("%.17g:  ", a, "  ", b, "  ", c, "  ", d)

 setparam("REALFMT", "%y")
 writeln("%y:    ", a, "  ", b, "  ", c, "  ", d)

 setparam("REALFMT", "%j")
 writeln("%j:    ", a, "  ", b, "  ", c, "  ", d)
end-model
```

This model produces the following output.

```
Default: 1.23456789e+11  1.23456789e+11  12  12
%1.2f:   123456789000.00  123456789008.90  12.00  12.00
%.17g:   123456789000  123456789008.89999  12.0000000045  12
%y:      1.23456789e11  1.234567890089e11  1.20000000045e1  1.2e1
%j:      123456789000  123456789008.9  12.0000000045  12
```

That is, with the default printing format it is not possible to distinguish between `a` and `b` or to see that `c` is not an integer. All these numbers are output with their exact values when a sufficiently large number of decimals is specified or by using one of the formats `%j` or `%y` that produce exact floating point representations of minimal length.

CHAPTER 11

More about Integer Programming

This chapter presents two applications to (Mixed) Integer Programming of the programming facilities in Mosel that have been introduced in the previous chapters.

11.1 Cut generation

Cutting plane methods add constraints (cuts) to the problem that cut off parts of the convex hull of the integer solutions, thus drawing the solution of the LP relaxation closer to the integer feasible solutions and improving the bound provided by the solution of the relaxed problem.

The Xpress Optimizer provides automated cut generation (see the optimizer documentation for details). To show the effects of the cuts that are generated by our example we switch off the automated cut generation.

11.1.1 Example problem

The problem we want to solve is the following: a large company is planning to outsource the cleaning of its offices at the least cost. The NSITES office sites of the company are grouped into areas (set $\mathrm{AREAS} = \{1, ..., \mathrm{NAREAS}\}$). Several professional cleaning companies (set $\mathrm{CONTR} = \{1, ..., \mathrm{NCONTRACTORS}\}$) have submitted bids for the different sites, a cost of 0 in the data meaning that a contractor is not bidding for a site.

To avoid being dependent on a single contractor, adjacent areas have to be allocated to different contractors. Every site s (s in $\mathrm{SITES} = \{1, ..., \mathrm{NSITES}\}$) is to be allocated to a single contractor, but there may be between LOWCON_a and UPPCON_a contractors per area a.

11.1.2 Model formulation

For the mathematical formulation of the problem we introduce two sets of variables:

clean_{cs} indicates whether contractor c is cleaning site s
alloc_{ca} indicates whether contractor c is allocated any site in area a

The objective to minimize the total cost of all contracts is as follows (where PRICE_{sc} is the price per site and contractor):

$$\text{minimize} \sum_{c \in \mathrm{CONTR}} \sum_{s \in \mathrm{SITES}} \mathrm{PRICE}_{sc} \cdot \mathrm{clean}_{cs}$$

We need the following three sets of constraints to formulate the problem:

1. Each site must be cleaned by exactly one contractor.

$$\forall s \in \mathrm{SITES} : \sum_{c \in \mathrm{CONTR}} \mathrm{clean}_{cs} = 1$$

2. Adjacent areas must not be allocated to the same contractor.

$$\forall c \in \text{CONTR}, a, b \in \text{AREAS}, a > b \text{ and } \text{ADJACENT}_{ab} = 1: \quad \text{alloc}_{ca} + \text{alloc}_{cb} \leqslant 1$$

3. The lower and upper limits on the number of contractors per area must be respected.

$$\forall a \in \text{AREAS}: \sum_{c \in \text{CONTR}} \text{alloc}_{ca} \geqslant \text{LOWCON}_a$$
$$\forall a \in \text{AREAS}: \sum_{c \in \text{CONTR}} \text{alloc}_{ca} \leqslant \text{UPPCON}_a$$

To express the relation between the two sets of variables we need more constraints: a contractor c is allocated to an area a if and only if he is allocated a site s in this area, that is, y_{ca} is 1 if and only if some x_{cs} (for a site s in area a) is 1. This equivalence is expressed by the following two sets of constraints, one for each sense of the implication (AREA_s is the area a site s belongs to and NUMSITE_a the number of sites in area a):

$$\forall c \in \text{CONTR}, a \in \text{AREAS}: \text{alloc}_{ca} \leqslant \sum_{\substack{s \in \text{SITES} \\ \text{AREA}_s = a}} \text{clean}_{cs}$$

$$\forall c \in \text{CONTR}, a \in \text{AREAS}: \text{alloc}_{ca} \geqslant \frac{1}{\text{NUMSITE}_a} \cdot \sum_{\substack{s \in \text{SITES} \\ \text{AREA}_s = a}} \text{clean}_{cs}$$

11.1.3 Implementation

The resulting Mosel program is the following model `clean.mos`. The variables clean_{cs} are defined as a *dynamic array* and are only created if contractor c bids for site s (that is, $\text{PRICE}_{sc} > 0$ or, taking into account inaccuracies in the data, $\text{PRICE}_{sc} > 0.01$).

Another implementation detail that the reader may notice is the separate initialization of the array sizes: we are thus able to create all arrays with fixed sizes (with the exception of the previously mentioned array of variables that is explicitly declared dynamic). This allows Mosel to handle them in a more efficient way.

```
model "Office cleaning"
 uses "mmxprs","mmsystem"

 declarations
   PARAM: array(1..3) of integer
 end-declarations

 initializations from 'clparam.dat'
   PARAM
 end-initializations

 declarations
   NSITES = PARAM(1)                    ! Number of sites
   NAREAS = PARAM(2)                    ! Number of areas (subsets of sites)
   NCONTRACTORS = PARAM(3)              ! Number of contractors
   AREAS = 1..NAREAS
   CONTR = 1..NCONTRACTORS
   SITES = 1..NSITES
   AREA: array(SITES) of integer        ! Area site is in
   NUMSITE: array(AREAS) of integer     ! Number of sites in an area
   LOWCON: array(AREAS) of integer      ! Lower limit on the number of
                                        ! contractors per area
   UPPCON: array(AREAS) of integer      ! Upper limit on the number of
                                        ! contractors per area
   ADJACENT: array(AREAS,AREAS) of integer    ! 1 if areas adjacent, 0 otherwise
```

```
  PRICE: array(SITES,CONTR) of real    ! Price per contractor per site

  clean: dynamic array(CONTR,SITES) of mpvar ! 1 iff contractor c cleans site s
  alloc: array(CONTR,AREAS) of mpvar   ! 1 iff contractor allocated to a site
                                       !  in area a
end-declarations

initializations from 'cldata.dat'
  [NUMSITE,LOWCON,UPPCON] as 'AREA'
  ADJACENT
  PRICE
end-initializations

ct:=1
forall(a in AREAS) do
  forall(s in ct..ct+NUMSITE(a)-1)
    AREA(s):=a
  ct+= NUMSITE(a)
end-do

forall(c in CONTR, s in SITES | PRICE(s,c) > 0.01) create(clean(c,s))

! Objective: Minimize total cost of all cleaning contracts
Cost:= sum(c in CONTR, s in SITES) PRICE(s,c)*clean(c,s)

! Each site must be cleaned by exactly one contractor
forall(s in SITES) sum(c in CONTR) clean(c,s) = 1

! Ban same contractor from serving adjacent areas
forall(c in CONTR, a,b in AREAS | a > b and ADJACENT(a,b) = 1)
  alloc(c,a) + alloc(c,b) <= 1

! Specify lower & upper limits on contracts per area
forall(a in AREAS | LOWCON(a)>0)
  sum(c in CONTR) alloc(c,a) >= LOWCON(a)
forall(a in AREAS | UPPCON(a)<NCONTRACTORS)
  sum(c in CONTR) alloc(c,a) <= UPPCON(a)

! Define alloc(c,a) to be 1 iff some clean(c,s)=1 for sites s in area a
forall(c in CONTR, a in AREAS) do
  alloc(c,a) <= sum(s in SITES| AREA(s)=a) clean(c,s)
  alloc(c,a) >= 1.0/NUMSITE(a) * sum(s in SITES| AREA(s)=a) clean(c,s)
end-do

forall(c in CONTR) do
  forall(s in SITES) clean(c,s) is_binary
  forall(a in AREAS) alloc(c,a) is_binary
end-do

minimize(Cost)                 ! Solve the MIP problem
end-model
```

In the preceding model, we have chosen to implement the constraints that force the variables alloc_{ca} to become 1 whenever a variable clean_{cs} is 1 for some site s in area a in an aggregated way (this type of constraint is usually referred to as Multiple Variable Lower Bound, MVLB, constraints). Instead of

```
forall(c in CONTR, a in AREAS)
  alloc(c,a) >= 1.0/NUMSITE(a) * sum(s in SITES| AREA(s)=a) clean(c,s)
```

we could also have used the stronger formulation

```
forall(c in CONTR, s in SITES)
  alloc(c,AREA(s)) >= clean(c,s)
```

but this considerably increases the total number of constraints. The aggregated constraints are sufficient to express this problem, but this formulation is very loose, with the consequence that the

solution of the LP relaxation only provides a very bad approximation of the integer solution that we want to obtain. For large data sets the Branch-and-Bound search may therefore take a long time.

11.1.4 Cut-and-Branch

To improve this situation without blindly adding many unnecessary constraints, we implement a cut generation loop at the top node of the search that only adds those constraints that are violated be the current LP solution.

The cut generation loop (procedure `top_cut_gen`) performs the following steps:

- solve the LP and save the basis
- get the solution values
- identify violated constraints and add them to the problem
- load the modified problem and load the previous basis

```
procedure top_cut_gen
 declarations
   MAXCUTS = 2500              ! Max no. of constraints added in total
   MAXPCUTS = 1000             ! Max no. of constraints added per pass
   MAXPASS  = 50               ! Max no. passes
   ncut, npass, npcut: integer       ! Counters for cuts and passes
   feastol: real                     ! Zero tolerance
   solc: array(CONTR,SITES) of real  ! Sol. values for variables  `clean'
   sola: array(CONTR,AREAS) of real  ! Sol. values for variables  `alloc'
   objval,starttime: real
   cut: array(range) of linctr
   bas: basis                        ! LP basis
 end-declarations

 starttime:=gettime
 setparam("XPRS_CUTSTRATEGY", 0)     ! Disable automatic cuts
 setparam("XPRS_PRESOLVE", 0)        ! Switch presolve off
 feastol:= getparam("XPRS_FEASTOL")  ! Get the solver zero tolerance
 setparam("ZEROTOL", feastol * 10)   ! Set the comparison tolerance of Mosel
 ncut:=0
 npass:=0

 while (ncut<MAXCUTS and npass<MAXPASS) do
   npass+=1
   npcut:= 0
   minimize(XPRS_LIN, Cost)          ! Solve the LP
   if npass>1 and objval=getobjval: break
   savebasis(bas)                    ! Save the current basis
   objval:= getobjval                ! Get the objective value

   forall(c in CONTR) do             ! Get the solution values
     forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
     forall(s in SITES) solc(c,s):=getsol(clean(c,s))
   end-do

! Search for violated constraints and add them to the problem:
   forall(c in CONTR, s in SITES)
     if solc(c,s) > sola(c,AREA(s)) then
       cut(ncut):= alloc(c,AREA(s)) >= clean(c,s)
       ncut+=1
       npcut+=1
       if npcut>MAXPCUTS or ncut>MAXCUTS: break 2
     end-if

   writeln("Pass ", npass, " (", gettime-starttime, " sec), objective value ",
           objval, ", cuts added: ", npcut, " (total ", ncut,")")
```

```
    if npcut=0 then
      break
    else
      loadprob(Cost)                  ! Reload the problem
      loadbasis(bas)                  ! Load the saved basis
    end-if
  end-do
                                      ! Display cut generation status
  write("Cut phase completed: ")
  if (ncut >= MAXCUTS) then writeln("space for cuts exhausted")
  elif (npass >= MAXPASS) then writeln("maximum number of passes reached")
  else writeln("no more violations or no improvement to objective")
  end-if
end-procedure
```

Assuming we add the definition of procedure `top_cut_gen` to the end of our model, we need to add its declaration at the beginning of the model:

```
forward procedure topcutgen
```

and the call to this function immediately before the optimization:

```
top_cut_gen                           ! Constraint generation at top node
minimize(Cost)                        ! Solve the MIP problem
```

Since we wish to use our own cut strategy, we switch off the default cut generation in Xpress Optimizer:

```
setparam("XPRS_CUTSTRATEGY", 0)
```

We also turn the presolve off since we wish to access the solution to the original problem after solving the LP-relaxations:

```
setparam("XPRS_PRESOLVE", 0)
```

11.1.5 Comparison tolerance

In addition to the parameter settings we also retrieve the feasibility tolerance used by Xpress Optimizer: the solver works with tolerance values for integer feasibility and solution feasibility that are typically of the order of 10^{-6} by default. When evaluating a solution, for instance by performing comparisons, it is important to take into account these tolerances.

After retrieving the feasibility tolerance of the solver we set the comparison tolerance of Mosel (`ZEROTOL`) to this value. This means, for example, the test $x = 0$ evaluates to true if x lies between *-ZEROTOL* and *ZEROTOL*, $x \leqslant 0$ is true if the value of x is at most *ZEROTOL*, and $x > 0$ is fulfilled if x is greater than *ZEROTOL*.

Comparisons in Mosel always use a tolerance, with a very small default value. By resetting this parameter to the solver feasibility tolerance Mosel evaluates solution values just like Xpress Optimizer.

11.1.6 Branch-and-Cut

The cut generation loop presented in the previous subsection only generates violated inqualities at the top node before entering the Branch-and-Bound search and adds them to the problem in the form of additional constraints. We may do the same using the *cut manager* of Xpress Optimizer. In this case, the violated constraints are added to the problem via the *cut pool*. We may even generate and add cuts during the Branch-and-Bound search. A cut added at a node using `addcuts` only applies to this node and its descendants, so one may use this functionality to define *local cuts* (however, in our example, all generated cuts are valid globally).

The cut manager is set up with a call to procedure `tree_cut_gen` before starting the optimization (preceded by the declaration of the procedure using `forward` earlier in the program). To avoid initializing the solution arrays and the feasibility tolerance repeatedly, we now turn these into globally

defined objects:

```
declarations
  feastol: real                         ! Zero tolerance
  solc: array(CONTR,SITES) of real      ! Sol. values for variables  `clean'
  sola: array(CONTR,AREAS) of real      ! Sol. values for variables  `alloc'
end-declarations

tree_cut_gen                            ! Set up cut generation in B&B tree
minimize(Cost)                          ! Solve the MIP problem
```

As we have seen before, procedure `tree_cut_gen` disables the default cut generation and turns presolve off. It also indicates the number of extra rows to be reserved in the matrix for the cuts we are generating:

```
procedure tree_cut_gen
  setparam("XPRS_CUTSTRATEGY", 0)       ! Disable automatic cuts
  setparam("XPRS_PRESOLVE", 0)          ! Switch presolve off
  setparam("XPRS_EXTRAROWS", 5000)      ! Reserve extra rows in matrix

  feastol:= getparam("XPRS_FEASTOL")    ! Get the zero tolerance
  setparam("zerotol", feastol * 10)     ! Set the comparison tolerance of Mosel

  setcallback(XPRS_CB_OPTNODE, ->cb_node)
end-procedure
```

The last line of this procedure defines the *optimal node (optnode) callback* function that will be called by the optimizer from every node of the Branch-and-Bound search tree if the node LP relaxation has been solved to optimality. This cut generation routine (function `cb_node`) performs the following steps:

- get the solution values
- identify violated inequalities and add them to the problem

It is implemented as follows (we restrict the generation of cuts to the first three levels, *i.e.* `depth <4`, of the search tree):

```
function cb_node:boolean

 declarations
   ncut: integer                            ! Counters for cuts
   cut: dynamic array(range) of linctr      ! Cuts
   cutid: dynamic array(range) of integer   ! Cut type identification
   type: dynamic array(range) of integer    ! Cut constraint type
 end-declarations

 depth:=getparam("XPRS_NODEDEPTH")
 node:=getparam("XPRS_NODES")

 if depth<4 then
   ncut:=0

 ! Get the solution values
   forall(c in CONTR) do
     forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
     forall(s in SITES) solc(c,s):=getsol(clean(c,s))
   end-do

 ! Search for violated constraints
   forall(c in CONTR, s in SITES | solc(c,s) > sola(c,AREA(s))) do
     cut(ncut):= alloc(c,AREA(s)) - clean(c,s)
     cutid(ncut):= 1
     type(ncut):= CT_GEQ
     ncut+=1
```

```
    end-do

  ! Add cuts to the problem
    if ncut>0 then
      addcuts(cutid, type, cut);
      writeln("Cuts added : ", ncut, " (depth ", depth, ", node ", node,
              ", obj. ", getparam("XPRS_LPOBJVAL"), ")")
    end-if
  end-if

end-function
```

The prototype of this function is prescribed by the type of the callback (see the Xpress Optimizer Reference Manual and the documentation of `setcallback` in the chapter on *mmxprs* in the Mosel Language Reference Manual). At every node this function is called repeatedly, followed by a re-solution of the current LP.

An alternative syntax for defining callbacks relies on the subroutine name: with this syntax the function implementing the callback must be declared as `public` in order to make its name accessible to the external program that invokes the callback:

```
 forward public function cb_node:boolean
...
 setcallback(XPRS_CB_OPTNODE, "cb_node")
```

11.2 Column generation

The technique of column generation is used for solving linear problems with a huge number of variables for which it is not possible to generate explicitly all columns of the problem matrix. Starting with a very restricted set of columns, after each solution of the problem a column generation algorithm adds one or several columns that improve the current solution. These columns must have a negative reduced cost (in a minimization problem) and are calculated based on the dual value of the current solution.

For solving large MIP problems, column generation typically has to be combined with a Branch-and-Bound search, leading to a so-called Branch-and-Price algorithm. The example problem described below is solved by solving a sequence of LPs without starting a tree search.

11.2.1 Example problem

A paper mill produces rolls of paper of a fixed width $MAXWIDTH$ that are subsequently cut into smaller rolls according to the customer orders. The rolls can be cut into $NWIDTHS$ different sizes. The orders are given as demands for each width i ($DEMAND_i$). The objective of the paper mill is to satisfy the demand with the smallest possible number of paper rolls in order to minimize the losses.

11.2.2 Model formulation

The objective of minimizing the total number of rolls can be expressed as choosing the best set of cutting patterns for the current set of demands. Since it may not be obvious how to calculate all possible cutting patterns by hand, we start off with a basic set of patterns ($PATTERNS_1$,..., $PATTERNS_{NWIDTH}$), that consists of cutting small rolls all of the same width as many times as possible (and at most the demanded quantity) out of the large roll. This type of problem is called a *cutting stock problem*.

If we define variables use_j to denote the number of times a cutting pattern j ($j \in WIDTHS = \{1, ..., NWIDTH\}$) is used, then the objective becomes to minimize the sum of these

variables, subject to the constraints that the demand for every size has to be met.

$$\text{minimize} \sum_{j \in WIDTHS} use_j$$

$$\sum_{j \in WIDTHS} PATTERNS_{ij} \cdot use_j \geqslant DEMAND_i$$

$$\forall j \in WIDTHS : use_j \leqslant \text{ceil}(DEMAND_j / PATTERNS_{jj}),\ use_j \in \mathbf{N}$$

Function $ceil$ means rounding to the next larger integer value.

11.2.3 Implementation

The first part of the Mosel model `paper.mos` implementing this problem looks as follows:

```
model Papermill
 uses "mmxprs"

 forward procedure columngen
 forward function knapsack(C:array(range) of real, A:array(range) of real,
                           B:real, D:array(range) of integer,
                           xbest:array(range) of integer,
                           pass: integer): real
 forward procedure shownewpat(dj:real, vx: array(range) of integer)

 declarations
   NWIDTHS = 5                            ! Number of different widths
   WIDTHS = 1..NWIDTHS                    ! Range of widths
   RP: range                              ! Range of cutting patterns
   MAXWIDTH = 94                          ! Maximum roll width
   EPS = 1e-6                             ! Zero tolerance

   WIDTH: array(WIDTHS) of real           ! Possible widths
   DEMAND: array(WIDTHS) of integer       ! Demand per width
   PATTERNS: array(WIDTHS,WIDTHS) of integer ! (Basic) cutting patterns

   use: dynamic array(RP) of mpvar        ! Rolls per pattern
   soluse: dynamic array(RP) of real      ! Solution values for variables  `use'
   Dem: array(WIDTHS) of linctr           ! Demand constraints
   MinRolls: linctr                       ! Objective function

   KnapCtr, KnapObj: linctr               ! Knapsack constraint+objective
   x: array(WIDTHS) of mpvar              ! Knapsack variables
 end-declarations

 WIDTH::  [ 17, 21, 22.5,  24, 29.5]
 DEMAND:: [150, 96,   48, 108,  227]

 forall(j in WIDTHS)                      ! Make basic patterns
   PATTERNS(j,j) := minlist(floor(MAXWIDTH/WIDTH(j)),DEMAND(j))

 forall(j in WIDTHS) do
   create(use(j))                         ! Create NWIDTHS variables  `use'
   use(j) is_integer                      ! Variables are integer and bounded
   use(j) <= integer(ceil(DEMAND(j)/PATTERNS(j,j)))
 end-do

 MinRolls:= sum(j in WIDTHS) use(j)       ! Objective: minimize no. of rolls

 forall(i in WIDTHS)                      ! Satisfy all demands
   Dem(i):= sum(j in WIDTHS) PATTERNS(i,j) * use(j) >= DEMAND(i)

 columngen                                ! Column generation at top node

 minimize(MinRolls)                       ! Compute the best integer solution
                                          ! for the current problem (including
```

```
                                        ! the new columns)
writeln("Best integer solution: ", getobjval, " rolls")
write("  Rolls per pattern: ")
forall(i in RP) write(getsol(use(i)),", ")
```

The paper mill can satisfy the demand with just the basic set of cutting patterns, but it is likely to incur significant losses through wasting more than necessary of every large roll and by cutting more small rolls than its customers have ordered. We therefore employ a column generation heuristic to find more suitable cutting patterns.

The following procedure `columngen` defines a column generation loop that is executed at the top node (this heuristic was suggested by M. Savelsbergh for solving a similar cutting stock problem). The column generation loop performs the following steps:

- solve the LP and save the basis
- get the solution values
- compute a more profitable cutting pattern based on the current solution
- generate a new column (= cutting pattern): add a term to the objective function and to the corresponding demand constraints
- load the modified problem and load the saved basis

To be able to increase the number of variables $\mathrm{use_j}$ in this function, these variables have been declared at the beginning of the program as a *dynamic array* without specifying any index range.

By setting Mosel's comparison tolerance to *EPS*, the test $\mathrm{zbest} = 0$ checks whether zbest lies within *EPS* of 0 (see explanation in Section 11.1).

Switching off presolve for the column generation problem generally helps to improve performance when iteratively resolving the problem after adding a new column and warm-starting it with the previous basis.

```
procedure columngen
  declarations
    dualdem: array(WIDTHS) of real
    xbest: array(WIDTHS) of integer
    dw, zbest, objval: real
    bas: basis
  end-declarations

  setparam("XPRS_PRESOLVE", 0)          ! Switch presolve off
  setparam("zerotol", EPS)              ! Set comparison tolerance of Mosel
  npatt:=NWIDTHS
  npass:=1

  while(true) do
    minimize(XPRS_LIN, MinRolls)        ! Solve the LP

    savebasis(bas)                      ! Save the current basis
    objval:= getobjval                  ! Get the objective value

                                        ! Get the solution values
    forall(j in 1..npatt)  soluse(j):=getsol(use(j))
    forall(i in WIDTHS)  dualdem(i):=getdual(Dem(i))
                                        ! Solve a knapsack problem
    zbest:= knapsack(dualdem, WIDTH, MAXWIDTH, DEMAND, xbest, npass) - 1.0

    write("Pass ", npass, ": ")
    if zbest = 0 then
      writeln("no profitable column found.\n")
```

```
      break
    else
      shownewpat(zbest, xbest)          ! Print the new pattern
      npatt+=1
      create(use(npatt))                ! Create a new var. for this pattern
      use(npatt) is_integer

      MinRolls+= use(npatt)             ! Add new var. to the objective
      dw:=0
      forall(i in WIDTHS)
        if xbest(i) > 0 then
          Dem(i)+= xbest(i)*use(npatt)  ! Add new var. to demand constr.s
          dw:= maxlist(dw, ceil(DEMAND(i)/xbest(i) ))
        end-if
      use(npatt) <= dw                  ! Set upper bound on the new var.

      loadprob(MinRolls)                ! Reload the problem
      loadbasis(bas)                    ! Load the saved basis
    end-if
    npass+=1
  end-do

  writeln("Solution after column generation: ", objval, " rolls, ",
          getsize(RP), " patterns")
  write("   Rolls per pattern: ")
  forall(i in RP) write(soluse(i),", ")
  writeln

  setparam("XPRS_PRESOLVE", 1)         ! Switch presolve on

end-procedure
```

The preceding procedure `columngen` calls the following auxiliary function `knapsack` to solve an *integer knapsack problem* of the form

$$\text{maximize } z = \sum_{j \in WIDTHS} C_j \cdot x_j$$

$$\sum_{j \in WIDTHS} A_j \cdot x_j \leqslant B$$

$$\forall j \in WIDTHS : x_j \text{ integer}$$

$$\forall j \in WIDTHS : x_j \leqslant D_j$$

The function `knapsack` solves a second optimization problem that is independent of the main, cutting stock problem since the two have no variables in common. We thus effectively work with *two* problems in a single Mosel model.

For efficiency reasons we have defined the knapsack variables and constraints globally. The integrality condition on the knapsack variables remains unchanged between several calls to this function, so we establish it when solving the first knapsack problem. On the other hand, the knapsack constraint and the objective function have different coefficients at every execution, so we need to replace them every time the function is called.

We *reset* the knapsack constraints to 0 at the end of this function so that they do not unnecessarily increase the size of the main problem. The same is true in the other sense: *hiding* the demand constraints while solving the knapsack problem makes life easier for the optimizer, but is not essential for getting the correct solution.

```
function knapsack(C:array(range) of real, A:array(range) of real, B:real,
                  D:array(range) of integer, xbest:array(range) of integer,
                  pass: integer):real

! Hide the demand constraints
  forall(j in WIDTHS) sethidden(Dem(j), true)
```

```
! Define the knapsack problem
   KnapCtr := sum(j in WIDTHS) A(j)*x(j) <= B
   KnapObj := sum(j in WIDTHS) C(j)*x(j)

! Integrality condition
   if(pass=1) then
     forall(j in WIDTHS) x(j) is_integer
     forall(j in WIDTHS) x(j) <= D(j)
   end-if

   maximize(KnapObj)
   returned:=getobjval
   forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

! Reset knapsack constraint and objective, unhide demand constraints
   KnapCtr := 0
   KnapObj := 0
   forall(j in WIDTHS) sethidden(Dem(j), false)
 end-function
```

To complete the model, we add the following procedure `shownewpat` to print every new pattern we find.

```
 procedure shownewpat(dj:real, vx: array(range) of integer)
   declarations
     dw: real
   end-declarations

   writeln("new pattern found with marginal cost ", dj)
   write("   Widths distribution: ")
   dw:=0
   forall(i in WIDTHS) do
      write(WIDTH(i), ":", vx(i), "  ")
      dw += WIDTH(i)*vx(i)
   end-do
   writeln("Total width: ", dw)
 end-procedure

end-model
```

11.2.4 Alternative implementation: Working with multiple problems

The implementation of the function `knapsack` in the previous section uses the `sethidden` functionality to blend out parts of the problem definition. The two parts of the problem (the main cutting stock problem and the problem solved in the `knapsack` routine) do not have any elements in common, that is, we really are solving two different problems within a single model.

An alternative implementation (available since Mosel 3.0) consists in formulating this model as two separate problems within the same model file.

The implementation as two separate problems in the model file `papers.mos` requires only few changes to the previous model formulation:

1. The declaration of a subproblem 'Knapsack' is added to the global declarations at the start of the model definition.

```
      declarations
       Knapsack: mpproblem                ! Knapsack subproblem
      end-declarations
```

2. The implementation of function `knapsack` now works within the subproblem 'Knapsack' instead

of hiding and unhiding subsets of the constraints. The scope of the subproblem is marked by the keywords `with mpproblem [do ... end-do]`.

```
function knapsack(C:array(range) of real, A:array(range) of real, B:real,
                  D:array(range) of integer, xbest:array(range) of integer,
                  pass: integer):real

  with Knapsack do

 ! Redefine the knapsack problem
    KnapCtr := sum(j in WIDTHS) A(j)*x(j) <= B
    KnapObj := sum(j in WIDTHS) C(j)*x(j)

 ! Integrality condition
    if pass=1 then
      forall(j in WIDTHS) x(j) is_integer
      forall(j in WIDTHS) x(j) <= D(j)
    end-if

    maximize(KnapObj)
    returned:=getobjval
    forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

  end-do

end-function
```

CHAPTER 12

Extensions to Linear Programming

The two examples (recursion and Goal Programming) in this chapter show how Mosel can be used to implement extensions of Linear Programming.

12.1 Recursion

Recursion, more properly known as *Successive Linear Programming (SLP)*, is a technique whereby LP may be used to solve certain non-linear problems. Some coefficients in an LP problem are defined to be functions of the optimal values of LP variables. When an LP problem has been solved, the coefficients are re-evaluated and the LP re-solved. Under some assumptions this process may converge to a local (though not necessarily a global) optimum.

This section describes the complete implementation of a recursion algorithm with Mosel using an LP solver. Section 17.4.2 shows how the example problem discussed in this section can be implemented and solved in a significantly simpler manner with Xpress NonLinear using the SLP solver.

12.1.1 Example problem

Consider the following financial planning problem: We wish to determine the yearly interest rate x so that for a given set of payments we obtain the final balance of 0. Interest is paid quarterly according to the following formula:

$$\mathrm{interest}_t = (92/365) \cdot \mathrm{balance}_t \cdot \mathrm{interest_rate}$$

The balance at time t ($t = 1, \ldots, T$) results from the balance of the previous period $t-1$ and the net of payments and interest:

$$\mathrm{net}_t = \mathrm{Payments}_t - \mathrm{interest}_t$$
$$\mathrm{balance}_t = \mathrm{balance}_{t-1} - \mathrm{net}_t$$

12.1.2 Model formulation

This problem cannot be modeled just by LP because we have the T products

$$\mathrm{balance}_t \cdot \mathrm{interest_rate}$$

which are non-linear. To express an approximation of the original problem by LP we replace the interest rate variable x by a (constant) guess X of its value and a deviation variable dx

$$x = X + dx$$

The formula for the quarterly interest payment i_t therefore becomes

$$\begin{aligned} interest_t &= 92/365 \cdot (balance_{t-1} \cdot x) \\ &= 92/365 \cdot (balance_{t-1} \cdot (X + dx)) \\ &= 92/365 \cdot (balance_{t-1} \cdot X + balance_{t-1} \cdot dx) \end{aligned}$$

where $balance_t$ is the balance at the beginning of period t.

We now also replace the balance $balance_{t-1}$ in the product with dx by a guess B_{t-1} and a deviation db_{t-1}

$$\begin{aligned} iinterest_t &= 92/365 \cdot (balance_{t-1} \cdot X + (B_{t-1} + db_{t-1}) \cdot dx) \\ &= 92/365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx + db_{t-1} \cdot dx) \end{aligned}$$

which can be approximated by dropping the product of the deviation variables

$$interest_t = 92/365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx)$$

To ensure feasibility we add penalty variables $eplus_t$ and $eminus_t$ for positive and negative deviations in the formulation of the constraint:

$$interest_t = 92/365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx + eplus_t - eminus_t)$$

The objective of the problem is to get feasible, that is to minimize the deviations:

$$\text{minimize} \sum_{t \in QUARTERS} (eplus_t + eminus_t)$$

12.1.3 Implementation

The Mosel model (file `recurse.mos`) then looks as follows (note the balance variables $balance_t$ as well as the deviation dx and the quarterly nets net_t are defined as free variables, that is, they may take any values between minus and plus infinity):

```
model Recurse
 uses "mmxprs"

 forward procedure solverecurse

 declarations
   T=6                                ! Time horizon
   QUARTERS=1..T                      ! Range of time periods
   P,R,V: array(QUARTERS) of real     ! Payments
   B: array(QUARTERS) of real         ! Initial guess as to balances b(t)
   X: real                            ! Initial guess as to interest rate x

   interest: array(QUARTERS) of mpvar  ! Interest
   net: array(QUARTERS) of mpvar       ! Net
   balance: array(QUARTERS) of mpvar   ! Balance
   x: mpvar                            ! Interest rate
   dx: mpvar                           ! Change to x
   eplus, eminus: array(QUARTERS) of mpvar  ! + and - deviations
 end-declarations

 X:= 0.00
 B:: [1, 1, 1, 1, 1, 1]
 P:: [-1000, 0, 0, 0, 0, 0]
 R:: [206.6, 206.6, 206.6, 206.6, 206.6, 0]
 V:: [-2.95, 0, 0, 0, 0, 0]
                                    ! net = payments - interest
```

```
forall(t in QUARTERS) net(t) = (P(t)+R(t)+V(t)) - interest(t)

                                    ! Money balance across periods
forall(t in QUARTERS) balance(t) = if(t>1, balance(t-1), 0) - net(t)

forall(t in 2..T) Interest(t):=     ! Approximation of interest
  -(365/92)*interest(t) + X*balance(t-1) + B(t-1)*dx + eplus(t) - eminus(t) = 0

Def:= X + dx = x                    ! Define the interest rate: x = X + dx

Feas:= sum(t in QUARTERS) (eplus(t)+eminus(t))  ! Objective: get feasible

interest(1) = 0                     ! Initial interest is zero
forall (t in QUARTERS) net(t) is_free
forall (t in 1..T-1) balance(t) is_free
balance(T) = 0                      ! Final balance is zero
dx is_free

minimize(Feas)                      ! Solve the LP-problem

solverecurse                        ! Recursion loop

                                    ! Print the solution
writeln("\nThe interest rate is ", x.sol)
write(strfmt("t",5), strfmt(" ",4))
forall(t in QUARTERS) write(strfmt(t,5), strfmt(" ",3))
write("\nBalances ")
setparam("realfmt", "%8.2f")
forall(t in QUARTERS) write(balance(t).sol)
write("\nInterest ")
forall(t in QUARTERS) write(interest(t).sol)

end-model
```

In the model above we have declared the procedure `solverecurse` that executes the recursion but it has not yet been defined. The recursion on x and the $balance_t$ ($t = 1, ..., T-1$) is implemented by the following steps:

(a) The B_{t-1} in constraints $Interest_t$ get the prior solution value of $balance_{t-1}$
(b) The X in constraints $Interest_t$ get the prior solution value of x
(c) The X in constraint Def gets the prior solution value of x

We say we have *converged* when the change in dx ($variation$) is less than 0.000001 (*TOLERANCE*). By setting Mosel's comparison tolerance to this value the test $variation > 0$ checks whether $variation$ is greater than *TOLERANCE*.

```
procedure solverecurse
  declarations
    TOLERANCE=0.000001              ! Convergence tolerance
    variation: real                 ! Variation of x
    BC: array(QUARTERS) of real
    bas: basis                      ! LP basis
  end-declarations

  setparam("zerotol", TOLERANCE)    ! Set Mosel comparison tolerance
  variation:=1.0
  ct:=0

  while(variation>0) do
    savebasis(bas)                  ! Save the current basis
    ct+=1
    forall(t in 2..T)
      BC(t-1):= getsol(balance(t-1))  ! Get solution values for balance(t)'s
    XC:= getsol(x)                  ! and x
    write("Round ", ct, " x:", getsol(x), " (variation:", variation,"), ")
    writeln("Simplex iterations: ", getparam("XPRS_SIMPLEXITER"))
```

```
    forall(t in 2..T) do              ! Update coefficients
      Interest(t)+= (BC(t-1)-B(t-1))*dx
      B(t-1):=BC(t-1)
      Interest(t)+= (XC-X)*balance(t-1)
    end-do
    Def+= XC-X
    X:=XC
    oldxval:=XC                       ! Store solution value of x

    loadprob(Feas)                    ! Reload the problem into the optimizer
    loadbasis(bas)                    ! Reload previous basis
    minimize(Feas)                    ! Re-solve the LP-problem

    variation:=abs(getsol(x)-oldxval) ! Change in dx
  end-do
end-procedure
```

With the initial guesses 0 for X and 1 for all B_t the model converges to an interest rate of 5.94413% (x = 0.0594413).

12.2 Goal Programming

Goal Programming is an extension of Linear Programming in which targets are specified for a set of constraints. In Goal Programming there are two basic models: the pre-emptive (lexicographic) model and the Archimedian model. In the pre-emptive model, goals are ordered according to priorities. The goals at a certain priority level are considered to be infinitely more important than the goals at the next level. With the Archimedian model weights or penalties for not achieving targets must be specified, and we attempt to minimize the sum of the weighted infeasibilities.

If constraints are used to construct the goals, then the goals are to minimize the violation of the constraints. The goals are met when the constraints are satisfied.

The first example in this section demonstrates how Mosel can be used for implementing *pre-emptive Goal Programming with constraints*. We try to meet as many goals as possible, taking them in priority order.

When the goals are specified as objective functions the resulting problem is often refered to as a *multi-objective optimization problem*. Xpress Optimizer supports the specification of multiple objective functions along with some configuration options, within Mosel this functionality is accessible through the *mmxprs* module as shown in the second example in section.

12.2.1 Example problem

The objective is to solve a problem with two variables x and y ($x, y \geqslant 0$), the constraint

$$100 \cdot x + 60 \cdot y \leqslant 600$$

and the three goal constraints

$$\begin{aligned} \mathrm{Goal}_1&: 7 \cdot x + 3 \cdot y \geqslant 40 \\ \mathrm{Goal}_2&: 10 \cdot x + 5 \cdot y = 60 \\ \mathrm{Goal}_3&: 5 \cdot x + 4 \cdot y \geqslant 35 \end{aligned}$$

where the order given corresponds to their priorities.

12.2.2 Implementation

To increase readability, the implementation of the Mosel model (file `goalctrp.mos`) is organized into

three blocks: the problem is stated in the main part, procedure `preemptive` implements the solution strategy via preemptive Goal Programming, and procedure `printsol` produces a nice solution printout.

```
model GoalCtrPreempt
 uses "mmxprs", "mmsystem"

 forward procedure preemptive
 forward procedure printsol(i:integer)

 declarations
   NGOALS=3                            ! Number of goals
   GOALS=1..NGOALS                     ! Set of goals
   x,y: mpvar                          ! Decision variables
   dev: array(1..2*NGOALS) of mpvar    ! Deviation from goals
   MinDev: linctr                      ! Objective function
   Goal: array(GOALS) of linctr        ! Goal constraints
 end-declarations

 100*x + 60*y <= 600                   ! Define a constraint

! Define the goal constraints
 Goal(1):=  7*x + 3*y >= 40
 Goal(2):= 10*x + 5*y = 60
 Goal(3):=  5*x + 4*y >= 35

 preemptive                            ! Pre-emptive Goal Programming
```

At the end of the main part, we call procedure `preemptive` to solve this problem by pre-emptive Goal Programming. In this procedure, the goals are at first entirely removed from the problem ('hidden'). We then add them successively to the problem and re-solve it until the problem becomes infeasible, that is, the deviation variables forming the objective function are not all 0. Depending on the constraint type (obtained with `gettype`) of the goals, we need one (for inequalities) or two (for equalities) deviation variables.

Let us have a closer look at the first goal (Goal_1), a $\geqslant$ constraint: the left hand side (all terms with decision variables) must be at least 40 to satisfy the constraint. To ensure this, we add a dev_2. The goal constraint becomes $7 \cdot x + 3 \cdot y + dev_2 \geqslant 40$ and the objective function is to minimize dev_2. If this is feasible (0-valued objective), then we remove the deviation variable from the goal, thus turning it into a *hard constraint*. It is not required to remove it from the objective since minimization will always force this variable to take the value 0.

We then continue with Goal_2: since this is an equation, we need variables for positive and negative deviations, dev_3 and dev_4. We add $dev_4 - dev_3$ to the constraint, turning it into $10 \cdot x + 5 \cdot y + dev_4 - dev_3 = 60$ and the objective now is to minimize the absolute deviation $dev_4 + dev_3$. And so on.

```
 procedure preemptive

! Remove (=hide) goal constraint from the problem
   forall(i in GOALS) sethidden(Goal(i), true)

   i:=0
   while (i<NGOALS) do
     i+=1
     sethidden(Goal(i), false)        ! Add (=unhide) the next goal

     case gettype(Goal(i)) of         ! Add deviation variable(s)
       CT_GEQ: do
                 Deviation:= dev(2*i)
                 MinDev += Deviation
               end-do
       CT_LEQ: do
                 Deviation:= -dev(2*i-1)
```

```
                MinDev += dev(2*i-1)
              end-do
      CT_EQ : do
                Deviation:= dev(2*i) - dev(2*i-1)
                MinDev += dev(2*i) + dev(2*i-1)
              end-do
      else    writeln("Wrong constraint type")
              break
    end-case
    Goal(i)+= Deviation

    minimize(MinDev)                ! Solve the LP-problem
    writeln(" Solution(", i,"): x: ", x.sol, ", y: ", y.sol)

    if getobjval>0 then
      writeln("Cannot satisfy goal ", i)
      break
    end-if
    Goal(i)-= Deviation             ! Remove deviation variable(s) from goal
  end-do

  printsol(i)                       ! Solution printout
end-procedure
```

The procedure `sethidden(c:linctr, b:boolean)` has already been used in the previous chapter (Section 11.2) without giving any further explanation. With this procedure, constraints can be removed ('hidden') from the problem solved by the optimizer without deleting them in the problem definition. So effectively, the optimizer solves a *subproblem* of the problem originally stated in Mosel.

To complete the model, below is the procedure `printsol` for printing the results.

```
procedure printsol(i:integer)
  declarations
    STypes={CT_GEQ, CT_LEQ, CT_EQ}
    ATypes: array(STypes) of string
  end-declarations

  ATypes::([CT_GEQ, CT_LEQ, CT_EQ])[">=", "<=", "="]

  writeln(" Goal", textfmt("Target",11), textfmt("Value",7))
  forall(g in 1..i)
    writeln(formattext("%4d %3s %5g %7g", g, ATypes(Goal(g).type),
      -Goal(g).coeff, Goal(g).act-dev(2*g).sol+dev(2*g-1).sol))

  forall(g in GOALS)
    if dev(2*g).sol>0 then
      writeln(" Goal(",g,") deviation from target: -", dev(2*g).sol)
    elif dev(2*g-1).sol>0 then
      writeln(" Goal(",g,") deviation from target: +", dev(2*g-1).sol)
    end-if
end-procedure

end-model
```

When running the program, one finds that the first two goals can be satisfied, but not the third.

12.2.3 Multi-objective optimization with Xpress Optimizer

The `minimize`/`maximize` routines of *mmxprs* have additional forms that allow the user to specify multiple objective functions along with configuration settings for each objective. The `objconfig` constructor used in the example below takes as its arguments the priority (all equal values to indicate the Archimedian case, otherwise the pre-emptive case applies where objectives with higher priority values will be considered first), a weight value (negative values inverse the sense of the objective), followed by absolute or relative tolerance values to be applied in the pre-emptive case when transforming an

objective into a constraint after a solution has been determined for it.

Specific solution reporting functionality for multi-objective solves includes the number of objectives for which the problem has been solved `XPRS_SOLVEDOBJS` (this value may be lower than the total number of specified objectives if the final problems status is 'infeasible') and the possibility to access summary information about the iterations of the pre-emptive solving via `getobjrealattr` or `getobjintattr`.

```
model GoalObjPreempt
 uses "mmxprs", "mmsystem"

 declarations
   NGOALS=3                           ! Number of goals
   GOALS=1..NGOALS                    ! Set of goals
   x,y: mpvar                         ! Variables
   Goal: array(GOALS) of linctr       ! Goal objective expressions
   ObjCfg: array(GOALS) of objconfig  ! Goal configuration
 end-declarations

 Limit:= 42*x + 13*y <= 100           ! Define a constraint

! Define the goal objectives
 Goal(1):=   5*x +  2*y - 20
 Goal(2):=  -3*x + 15*y - 48
 Goal(3):= 1.5*x + 21*y - 3.8

! Configuration of the goal objectives
 ObjCfg(1):= objconfig(3, -1, 0, 0.1)  ! 1st, maximize, 10% tolerance
 ObjCfg(2):= objconfig(2,  1, 4, 0)    ! 2nd, minimize, absolute tolerance of 4
 ObjCfg(3):= objconfig(1, -1, 0, 0.2)  ! 3rd, maximize, 20% tolerance

 minimize(Goal,ObjCfg)                ! Solve the multi-objective problem

! Solution reporting
 writeln("Solved for ", getparam("XPRS_SOLVEDOBJS"), " goals")

! Solution values in iterations of pre-emptive solving
 forall(i in GOALS)
   writeln(" Iteration ", i, ": val=", getobjrealattr(i, "XPRS_LPOBJVAL"))

! Final solution
 writeln(" Solution: x: ", x.sol, ", y: ", y.sol)
 writeln(" Goal", textfmt("Target",9), textfmt("Value",12))
 forall(g in GOALS)
   writeln(formattext("%4d %8s %12.6f", g, if(ObjCfg(g).weight=1,"min","max"),
      Goal(g).sol))
end-model
```

III. Working with the Mosel libraries

Overview

Whilst the two previous parts have shown how to work with the Mosel Language, this part introduces the programming language interface of Mosel in the form of the *Mosel C libraries*. The C interface is provided in the form of two libraries; it may especially be of interest to users who

- want to integrate models and/or solution algorithms written with Mosel into some larger system
- want to (re)use already existing parts of algorithms written in C
- want to interface Mosel with other software, for instance for graphically displaying results.

Other programming language interfaces available for Mosel are its *Java*, *.NET*, *C#*, *Python*, and *VBA* interfaces. They will be introduced with the help of small examples in Chapter 14.

All these programming language interfaces only enable the user to access models, but not to modify them. The latter is only possible with the *Mosel Native Interface*. Even more importantly, the Native Interface makes it possible to add new constants, types, and subroutines to the Mosel Language. For more detail the reader is referred to the Native Interface user guide that is provided as a separate document.

Chapter 13

C interface

This chapter gives an introduction to the C interface of Mosel. It shows how to execute models from C and how to access modeling objects from C. It is not possible to make changes to Mosel modeling objects from C using this interface, but the data and parameters used by a model may be modified via files or run time parameters.

13.1 Basic tasks

To work with a Mosel model, in the C language or with the command line interpreter, it always needs to be compiled, then loaded into Mosel and executed. In this section we show how to perform these basic tasks in C.

13.1.1 Compiling a model in C

The following example program shows how Mosel is initialized in C, and how a model file (extension `.mos`) is compiled into a **bi**nary **m**odel (BIM) file (extension `.bim`). To use the Mosel Model Compiler Library, we need to include the header file `xprm_mc.h` at the start of the C program.

For the sake of readability, in this program (file `ugcomp.c`), as for all others in this chapter, we only implement a rudimentary testing for errors.

```
#include <stdlib.h>
#include "xprm_mc.h"

int main()
{
 if(XPRMinit())                       /* Initialize Mosel */
  return 1;

 if(XPRMcompmod(NULL, "burglar2.mos", NULL, "Knapsack example"))
  return 2;                           /* Compile the model burglar2.mos,
                                         output the file burglar2.bim */
 XPRMfinish();                        /* Finish Mosel, clear everything */

 return 0;
}
```

The model `burglar2.mos` used here is the same as model `burglari.mos` in Section 2.1.3, but reading the data from file.

With version 1.4 of Mosel it becomes possible to redirect the BIM file that is generated by the compilation. Instead of writing it out to a physical file it may, for instance, be kept in memory or be written out in compressed format. The interested reader is refered to the whitepaper *Generalized file handling in Mosel*.

13.1.2 Executing a model in C

The example in this section shows how a Mosel binary model file (BIM) can be executed in C. The BIM file can of course be generated within the same program where it is executed, but here we leave out this step. A BIM file is an executable version of a model, but it does not include any data that is read in by the model from external files. It is portable, that is, it may be executed on a different type of architecture than the one it has been generated on. A BIM file produced by the Mosel compiler first needs to be loaded into Mosel (function `XPRMloadmod`) and can then be run by a call to function `XPRMrunmod`. To use these functions, we need to include the header file `xprm_rt.h` at the beginning of our program (named `ugrun.c`).

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 int result;

 if(XPRMinit())                        /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("burglar2.bim", NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))      /* Run the model */
  return 3;

 XPRMfinish();                         /* Finish Mosel, clear everything */

 return 0;
}
```

The compile/load/run sequence may also be performed with a single function call to `XPRMexecmod` (in this case we need to include the header file `xprm_mc.h`):

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
 int result;

 if(XPRMinit())                        /* Initialize Mosel */
  return 1;
                                       /* Execute = compile/load/run a model */
 if(XPRMexecmod(NULL, "burglar2.mos", NULL, &result, NULL))
  return 2;

 XPRMfinish();                         /* Finish Mosel, clear everything */

 return 0;
}
```

13.1.3 Termination

All program examples in this manual only serve to execute Mosel models. The corresponding model and Mosel itself are terminated (unloaded from memory) with the end of the C program. However, for embedding the execution of a Mosel model into some larger application it may be desirable to free the space used by the model or the execution of Mosel before the end of the application program. To this aim Mosel provides the functions `XPRMresetmod`, `XPRMunloadmod`, and `XPRMfinish`.

The function XPRMresetmod frees some resources allocated to a model, in particular (solution) data held in memory or temporary files that may have been created during its execution. The model remains loaded for later re-use. With a call to XPRMunloadmod a model is unloaded and all related resources are freed.
Function XPRMfinish performs the unloading of all models, frees all memory used by Mosel, and also removes the temporary directory/files that have been created by Mosel.

13.2 Parameters

In Part I the concept of parameters in Mosel has been introduced: when a Mosel model is executed from the command line, it is possible to pass new values for its parameters into the model. The same is possible with a model run in C. If, for instance, we want to run model 'Prime' from Section 8.3 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter LIMIT in the model), we may start a program with the following lines:

```
XPRMmodel mod;
int result;

if(XPRMinit())                                  /* Initialize Mosel */
 return 1;

if((mod=XPRMloadmod("prime.bim",NULL))==NULL)  /* Load a BIM file */
 return 2;

if(XPRMrunmod(mod,&result,"LIMIT=500"))        /* Run the model */
 return 3;
```

To use function XPRMexecmod instead of the compile/load/run sequence we have:

```
int result;

if(XPRMinit())                       /* Initialize Mosel */
 return 1;
                                     /* Execute with new parameter settings */
if(XPRMexecmod(NULL,"prime.mos","LIMIT=500",&result,NULL))
 return 2;
```

13.3 Accessing modeling objects and solution values

Using the Mosel libraries, it is not only possible to compile and run models, but also to access information on the different modeling objects.

13.3.1 Accessing sets

A complete version of a program (file ugparam1.c) for running the model 'Prime' mentioned in the previous section may look as follows (we work with a model prime2 that corresponds to the one printed in Section 8.3 but with all output printing removed because we are doing this in C, furthermore all entities accessed from C must be explicitly declared as public):

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue, setitem;
 XPRMset set;
```

```
 int result, type, i, size, first, last;

 if(XPRMinit())                              /* Initialize Mosel */
  return 1;

 if(XPRMexecmod(NULL, "prime2.mos", "LIMIT=500", &result, &mod))
  return 2;                                  /* Execute the model */

 type=XPRMfindident(mod, "SPrime", &rvalue); /* Get the object 'SPrime' */
 if((XPRM_TYP(type)!=XPRM_TYP_INT)||         /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_SET))          /* it must be a set of integers */
  return 3;
 set = rvalue.set;

 size = XPRMgetsetsize(set);                 /* Get the size of the set */
 if(size>0)
 {
  first = XPRMgetfirstsetndx(set);           /* Get number of the first index */
  last = XPRMgetlastsetndx(set);             /* Get number of the last index */
  printf("Prime numbers from 2 to %d:\n", LIM);
  for(i=first;i<=last;i++)                   /* Print all set elements */
   printf(" %d,", XPRMgetelsetval(set,i,&setitem)->integer);
  printf("\n");
 }

 XPRMfinish();                               /* Finish Mosel */

 return 0;
}
```

To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), we first retrieve the Mosel reference to this object using function `XPRMfindident`. We are then able to enumerate the elements of the set (using functions `XPRMgetfirstsetndx` and `XPRMgetlastsetndx`) and obtain their respective values with `XPRMgetelsetval`.

13.3.2 Retrieving solution values

The following program `ugsol1.c` executes the model 'Burglar3' (the same as model 'Burglar2' from Chapter 2 but with all output printing removed and all model entities declared as `public`) and prints out its solution.

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue, itemname;
 XPRMarray varr, darr;
 XPRMmpvar x;
 XPRMset set;
 int indices[1], result, type;
 double val;

 if(XPRMinit())                            /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("burglar3.bim", NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod, &result, NULL))        /* Run the model (includes
                                              optimization) */
  return 3;

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
```

```
 return 4;                                  /* Test whether a solution is found */

printf("Objective value: %g\n", XPRMgetobjval(mod));
                                            /* Print the obj. function value */

type=XPRMfindident(mod,"take",&rvalue);     /* Get the model object 'take' */
if((XPRM_TYP(type)!=XPRM_TYP_MPVAR)||       /* Check the type: */
   (XPRM_STR(type)!=XPRM_STR_ARR))          /* it must be an  `mpvar' array */
 return 5;
varr = rvalue.array;

type=XPRMfindident(mod,"VALUE",&rvalue);    /* Get the model object 'VALUE' */
if((XPRM_TYP(type)!=XPRM_TYP_REAL)||        /* Check the type: */
   (XPRM_STR(type)!=XPRM_STR_ARR))          /* it must be an array of reals */
 return 6;
darr = rvalue.array;

type=XPRMfindident(mod,"ITEMS",&rvalue);    /* Get the model object 'ITEMS' */
if((XPRM_TYP(type)!=XPRM_TYP_STRING)||      /* Check the type: */
   (XPRM_STR(type)!=XPRM_STR_SET))          /* it must be a set of strings */
 return 7;
set = rvalue.set;

XPRMgetfirstarrentry(varr, indices);        /* Get the first entry of array varr
                                               (we know that the array is dense
                                               and has a single dimension) */
do
{
 XPRMgetarrval(varr, indices, &x);          /* Get a variable from varr */
 XPRMgetarrval(darr, indices, &val);        /* Get the corresponding value */
 printf("take(%s): %g\t (item value: %g)\n", XPRMgetelsetval(set, indices[0],
        &itemname)->string, XPRMgetvsol(mod,x), val);
                                            /* Print the solution value */
} while(!XPRMgetnextarrentry(varr, indices));  /* Get the next index tuple */

XPRMfinish();                               /* Finish Mosel, clear everything */

 return 0;
}
```

The array of variables `varr` is enumerated using the array functions `XPRMgetfirstarrentry` and `XPRMgetnextarrentry`. These functions may be applied to arrays of any type and dimension (for higher numbers of dimensions, merely the size of the array `indices` that is used to store the index-tuples has to be adapted). With these functions we run systematically through all possible combinations of index tuples, hence the hint at *dense* arrays in the example. In the case of sparse arrays it is preferrable to use different enumeration functions that only enumerate those entries that are defined (see next section).

13.3.3 Sparse arrays

In Chapter 3 the problem 'Transport' has been introduced. The objective of this problem is to calculate the flows $\mathrm{flow}_{\mathrm{pr}}$ from a set of plants to a set of sales regions that satisfy all demand and supply constraints and minimize the total cost. Not all plants may deliver goods to all regions. The flow variables $\mathrm{flow}_{\mathrm{pr}}$ are therefore defined as a *sparse* array. The following example (file `ugarray1.c`) prints out all existing entries of the array of variables.

```
#include <stdio.h>
#include <stdlib.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue;
```

```
 XPRMarray varr;
 XPRMset *sets;
 int *indices, dim, result, type, i;

 if(XPRMinit())                          /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("transport.bim", NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod, &result, NULL))      /* Run the model */
  return 3;

 type=XPRMfindident(mod,"flow",&rvalue); /* Get the model object named 'flow' */
 if((XPRM_TYP(type)!=XPRM_TYP_MPVAR)||   /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_ARR))      /* it must be an array of unknowns */
  return 4;
 varr=rvalue.array;

 dim = XPRMgetarrdim(varr);              /* Get the number of dimensions of
                                            the array */
 indices = (int *)malloc(dim*sizeof(int));
 sets = (XPRMset *)malloc(dim*sizeof(XPRMset));

 XPRMgetarrsets(varr,sets);              /* Get the indexing sets  */
 XPRMgetfirstarrtruentry(varr,indices);  /* Get the first true index tuple */
 do
 {
  printf("flow(");
  for(i=0;i<dim-1;i++)
   printf("%s,",XPRMgetelsetval(sets[i],indices[i],&rvalue)->string);
  printf("%s), ",XPRMgetelsetval(sets[dim-1],indices[dim-1],&rvalue)->string);
 } while(!XPRMgetnextarrtruentry(varr,indices));  /* Get next true index tuple*/
 printf("\n");

 free(sets);
 free(indices);
 XPRMresetmod(mod);

 return 0;
}
```

In this example, we first get the number of indices (dimensions) of the array of variables `varr` (using function `XPRMgetarrdim`). We use this information to allocate space for the arrays `sets` and `indices` that will be used to store the indexing sets and single index tuples for this array respectively. We then read the indexing sets of the array (function `XPRMgetarrsets`) to be able to produce a nice printout.

The enumeration starts with the first defined index tuple, obtained with function `XPRMgetfirstarrtruentry`, and continues with a series of calls to `XPRMgetnextarrtruentry` until all defined entries have been enumerated.

At the end of the program example we have *reset* the model (using function `XPRMresetmod`), thus freeing some resources allocated to it, in particular deleting temporary files that may have been created during its execution.

13.4 Exchanging data between an application and a model

In the previous sections we have seen how to obtain solution information and other data from a Mosel model after its execution. For the integration of a model into an application a flow of information in the opposite sense, that is, from the host application to the model, will often also be required, in particular if data are generated by the application that serve as input to the model. It is possible to write out this

data to a (text) file or a database and read this file in from the model, but it is clearly more efficient to communicate such data in memory directly from the application to the model.

In this section we show two versions of our Burglar example where all input data is loaded from the application into the model, using dense and sparse data format respectively. The same communication mechanism, namely a combination of the two I/O drivers (see Section 17.1 for further detail) `raw` and `mem`, is also used to write back the solution from the model to the calling application.

An alternative communication mechanism is presented in Section 13.4.3. Instead of working with blocks of predefined size as in the previous cases, here data is passed through flows, allowing for dynamic sizing on the application level, a feature that is particularly useful for solution output with sparse data structures.

A separate example (Section 13.4.4) shows how to input and output *scalar data*.

13.4.1 Dense arrays

In the first instance we are going to consider a version of the 'Burglar' model that corresponds to the very first version we have seen in Section 2.1 where all arrays are indexed by the range set `ITEMS = 1..8`. In our C program `ugiodense.c` below, this corresponds to storing data in standard C arrays that are communicated to the Mosel model at the start of its execution.

```
#include <stdio.h>
#include "xprm_mc.h"

double vdata[8]={15,100,90,60,40,15,10, 1};   /* Input data: VALUE */
double wdata[8]={ 2, 20,20,30,40,30,60,10};   /* Input data: WEIGHT */
double solution[8];               /* Array for solution values */

int main()
{
 XPRMmodel mod;
 int i,result;
 char vdata_name[40];             /* File name of input data 'vdata' */
 char wdata_name[40];             /* File name of input data 'wdata' */
 char solution_name[40];          /* File name of solution values */
 char params[144];                /* Parameter string for model execution */

 if(XPRMinit())                   /* Initialize Mosel */
  return 1;

/* Prepare file names for 'initializations' using the 'raw' driver */
 sprintf(vdata_name, "noindex,mem:%p/%d", vdata, (int)sizeof(vdata));
 sprintf(wdata_name, "noindex,mem:%p/%d", wdata, (int)sizeof(wdata));
 sprintf(solution_name, "noindex,mem:%p/%d", solution, (int)sizeof(solution));

                                  /* Pass file names as execution param.s */
 sprintf(params, "VDATA='%s',WDATA='%s',SOL='%s'", vdata_name, wdata_name,
         solution_name);

 if(XPRMexecmod(NULL, "burglar6.mos", params, &result, &mod))
  return 2;                       /* Execute a model file */

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  return 3;                       /* Test whether a solution is found */

/* Display solution values obtained from the model */
 printf("Objective value: %g\n", XPRMgetobjval(mod));
 for(i=0;i<8;i++)
  printf(" take(%d): %g\n", i+1, solution[i]);

 XPRMresetmod(mod);               /* Reset the model */

 return 0;
}
```

In this example we use the `raw` I/O driver for communication between the application and the model it executes. Employing this driver means that data is saved in binary format. File names used with the `raw` driver have the form *`rawoption[,...],filename`*. The option `noindex` for this driver indicates that data is to be stored in *dense format*, that is, just the data entries without any information about the indices—this format supposes that the index set(s) is known in the Mosel model before data is read in. The *`filename`* uses the `mem` driver, this means that data is stored in memory. The actual location of the data is specified by giving the address of the corresponding memory block and its size.

The program above works with the following version of the 'Burglar' model where the locations of input and output data are specified by the calling application through model parameters. Instead of printing out the solution in the model, we copy the solution values of the decision variables `take` into the array of reals `soltake` that is written to memory and will be processed by the host application.

```
model Burglar6
 uses "mmxprs"

 parameters
  VDATA = ''; WDATA = ''         ! Locations of input data
  SOL = ''                       ! Location for solution data output
  WTMAX = 102                    ! Maximum weight allowed
 end-parameters

 declarations
  ITEMS = 1..8                   ! Index range for items

  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items

  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
  soltake: array(ITEMS) of real  ! Solution values
 end-declarations

 initializations from 'raw:'
  VALUE as VDATA  WEIGHT as WDATA
 end-initializations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the MIP-problem

! Output solution to calling application
 forall(i in ITEMS) soltake(i):= getsol(take(i))

 initializations to 'raw:'
  soltake as SOL
 end-initializations

end-model
```

13.4.2 Sparse arrays

Let us now take a look at the case where we use a set of strings instead of a simple range set to index the various arrays in our model. Storing the indices with the data values makes necessary slightly more complicated structures in our C program for the input and solution data. In the C program below (file `ugiosparse.c`), every input data entry defines both, the value and the weight coefficient for the corresponding index.

```
#include <stdio.h>
#include "xprm_mc.h"

const struct
{                                /* Initial values for array 'data': */
 const char *ind;                /*   index name */
 double val,wght;                /*   value and weight data entries */
} data[]={{"camera",15,2}, {"necklace",100,20}, {"vase",90,20},
          {"picture",60,30}, {"tv",40,40}, {"video",15,30},
          {"chest",10,60}, {"brick",1,10}};

struct
{                                /* Array to receive solution values: */
 const char *ind;                /*   index name */
 double val;                     /*   solution value */
} solution[8];

int main()
{
 XPRMmodel mod;
 int i,result;
 char data_name[40];             /* File name of input data 'data' */
 char solution_name[40];         /* File name of solution values */
 char params[96];                /* Parameter string for model execution */

 if(XPRMinit())                  /* Initialize Mosel */
  return 1;

/* Prepare file names for 'initializations' using the 'raw' driver */
 sprintf(data_name, "slength=0,mem:%p/%d", data, (int)sizeof(data));
 sprintf(solution_name, "slength=0,mem:%p/%d", solution, (int)sizeof(solution));

                                 /* Pass file names as execution param.s */
 sprintf(params, "DATA='%s',SOL='%s'", data_name, solution_name);

 if(XPRMexecmod(NULL, "burglar7.mos", params, &result, &mod))
  return 2;                      /* Execute a model file */

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  return 3;                      /* Test whether a solution is found */

/* Display solution values obtained from the model */
 printf("Objective value: %g\n", XPRMgetobjval(mod));
 for(i=0;i<8;i++)
  printf(" take(%s): %g\n", solution[i].ind, solution[i].val);

 XPRMresetmod(mod);

 return 0;
}
```

The use of the two I/O drivers is quite similar to what we have seen before. We now pass on data in *sparse format*, this means that every data entry is saved together with its index (tuple). Option `slength=0` of the `raw` driver indicates that strings are represented by pointers to null terminated arrays of characters (C-string) instead of fixed size arrays.

Similarly to the model of the previous section, the model `burglar7.mos` executed by the C program above reads and writes data from/to memory using the `raw` driver and the locations are specified by the calling application through the model parameters. Since the contents of the index set `ITEMS` is not defined in the model we have moved the declaration of the decision variables after the data input where the contents of the set is known, thus avoiding the creation of the array of decision variables as a dynamic array.

```
model Burglar7
 uses "mmxprs"
```

```
 parameters
  DATA = ''                          ! Location of input data
  SOL = ''                           ! Location for solution data output
  WTMAX = 102                        ! Maximum weight allowed
 end-parameters

 declarations
  ITEMS: set of string               ! Index set for items
  VALUE: array(ITEMS) of real        ! Value of items
  WEIGHT: array(ITEMS) of real       ! Weight of items
 end-declarations

 initializations from 'raw:'
  [VALUE,WEIGHT] as DATA
 end-initializations

 declarations
  take: array(ITEMS) of mpvar        ! 1 if we take item i; 0 otherwise
 end-declarations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                    ! Solve the MIP-problem

! Output solution to calling application
 forall(i in ITEMS) soltake(i):= getsol(take(i))

 initializations to 'raw:'
  soltake as SOL
 end-initializations

end-model
```

13.4.3 Dynamic data

The two examples of in-memory communication of dense and sparse data in the preceding sections have in commun that all data structures in the application, and in particular the structures to receive output data, are of fixed size. We therefore now introduce an alternative communication mechanism working with flows, that enables dynamic sizing of data structures on the application level, a feature that is particularly useful for solution output where effective data sizes are not known a priori. This communication mechanism is based on the callback I/O driver `cb` (see also Section 13.5). The main body of our C program now looks as follows.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xprm_mc.h"

                                  /* Input values for data: */
char *ind[]={"camera", "necklace", "vase", "picture", "tv", "video",
             "chest", "brick"};              /* Index names */
double vdata[]={15,100,90,60,40,15,10, 1};   /* Input data: VALUE */
double wdata[]={ 2, 20,20,30,40,30,60,10};   /* Input data: WEIGHT */
int datasize=8;

struct SolArray
{                                 /* Array to receive solution values: */
 const char *ind;                 /*   index name */
 double val;                      /*   solution value */
```

```
};

struct SolArray *solution;
int solsize;

int main()
{
 XPRMmodel mod;
 int i,result;
 char data_name[40];              /* File name of input data 'data' */
 char solution_name[40];          /* File name of solution values */
 char params[96];                 /* Parameter string for model execution */

 if(XPRMinit())                   /* Initialize Mosel */
  return 1;

/* Prepare file names for 'initializations' using the 'cb' driver */
 sprintf(data_name, "cb:%p", cbinit_from);
 sprintf(solution_name, "cb:%p", cbinit_to);

                                  /* Pass file names as execution param.s */
 sprintf(params, "DATAFILE='%s',SOLFILE='%s'", data_name, solution_name);

 if(XPRMexecmod(NULL, "burglar13.mos", params, &result, &mod))
  return 2;                       /* Execute a model file */

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  return 3;                       /* Test whether a solution is found */

/* Display solution values obtained from the model */
 printf("Objective value: %g\n", XPRMgetobjval(mod));
 for(i=0;i<solsize;i++)
  printf(" take(%s): %g\n", solution[i].ind, solution[i].val);

 XPRMresetmod(mod);

 return 0;
}
```

The function for *dynamic output retrieval* employs the Mosel library functions that we have already seen in Section 13.3 for models after their termination. The prototype of the function `cbinit_to` needs to be exactly as shown below.

```
int XPRM_RTC cbinit_to(XPRMcbinit cb, void *info, const char *label,
                       int type, XPRMalltypes *ref)
{
 XPRMarray solarr;
 XPRMset sets[1];
 int indices[1];
 XPRMalltypes rvalue;
 int ct;

 if(strcmp(label,"SOL")==0)
 {
  solarr=ref->array;

  solsize=XPRMgetarrsize(solarr);
  solution = (struct SolArray *)malloc(solsize * sizeof(struct SolArray));

  XPRMgetarrsets(solarr,sets);    /* Get the indexing sets
                                     (we know array has 1 dimension) */
  ct=0;
  XPRMgetfirstarrtruentry(solarr,indices); /* Get the first true index tuple */
  do
  {
   solution[ct].ind=XPRMgetelsetval(sets[0],indices[0],&rvalue)->string;
   XPRMgetarrval(solarr,indices,&rvalue);
   solution[ct].val=rvalue.real;
```

```
   ct++;
  } while(!XPRMgetnextarrtruentry(solarr,indices));
 }
 else
 {
  printf("Unknown output data item: %s %p\n", label, ref);
 }
 return 0;
}
```

The *dynamic data input* to a Mosel model uses a new set of dedicated library functions.

The format used to represent data is the same as the default text format used by `initializations` blocks. For example, the array definition

```
 mydata: [ ("ind1" 3) [5 1.2] ("ind2" 7) [4 6.5] ]
```

is represented by the following sequence of function calls:

```
XPRMcb_sendctrl(cb, XPRM_CBC_OPENLST, 0);     ! [
XPRMcb_sendctrl(cb, XPRM_CBC_OPENNDX, 0);     !   (
XPRMcb_sendstring(cb, "ind1", 0);             !     "ind1"
XPRMcb_sendint(cb, 3, 0);                     !     3
XPRMcb_sendctrl(cb, XPRM_CBC_CLOSENDX, 0);    !   )
XPRMcb_sendctrl(cb, XPRM_CBC_OPENLST, 0);     !   [
XPRMcb_sendint(cb, 5, 0);                     !     5
XPRMcb_sendreal(cb, 1.2, 0);                  !     1.2
XPRMcb_sendctrl(cb, XPRM_CBC_CLOSELST, 0);    !   ]
XPRMcb_sendctrl(cb, XPRM_CBC_OPENNDX, 0);     !   (
XPRMcb_sendstring(cb, "ind2", 0);             !     "ind2"
XPRMcb_sendint(cb, 7, 0);                     !      7
XPRMcb_sendctrl(cb, XPRM_CBC_CLOSENDX, 0);    !   )
XPRMcb_sendctrl(cb, XPRM_CBC_OPENLST, 0);     !   [
XPRMcb_sendint(cb, 4, 0);                     !     4
XPRMcb_sendreal(cb, 6.5, 0);                  !     6.5
XPRMcb_sendctrl(cb, XPRM_CBC_CLOSELST, 0);    !   ]
XPRMcb_sendctrl(cb, XPRM_CBC_CLOSELST, 0);    ! ]
```

The last argument '0' in these functions indicates that data is to be processed not immediately but only once the queue of tokens is full.

For our example, we thus have the following function definition (again, the prototype of the callback function must be defined exactly to the form expected by Mosel):

```
int XPRM_RTC cbinit_from(XPRMcbinit cb, void *info, const char *label,
                         int type, void *ref)
{
 int i;

 if(strcmp(label,"DATA")==0)
 {
  XPRMcb_sendctrl(cb, XPRM_CBC_OPENLST, 0);
  for(i=0;i<datasize;i++)
  {
   XPRMcb_sendctrl(cb, XPRM_CBC_OPENNDX, 0);
    XPRMcb_sendstring(cb, ind[i], -1, 0);
   XPRMcb_sendctrl(cb, XPRM_CBC_CLOSENDX, 0);
   XPRMcb_sendctrl(cb, XPRM_CBC_OPENLST, 0);
    XPRMcb_sendreal(cb, vdata[i], 0);
    XPRMcb_sendreal(cb, wdata[i], 0);
   XPRMcb_sendctrl(cb, XPRM_CBC_CLOSELST, 0);
  }
  XPRMcb_sendctrl(cb, XPRM_CBC_CLOSELST, 0);
  return 0;
 }
 else
```

```
 {
  fprintf(stderr,"Label  `%s' not found.\n",label);
  return 1;
 }
}
```

The model file `burglar13.mos` receives through its run-time parameters the callback functions that are to be used for data input/output in the `initializations` sections. The definition of the mathematical model is the same as in the previous model version and left out in the listing below.

```
model Burglar13
 uses "mmxprs"

 parameters
  DATAFILE = ''                    ! Location of input data
  SOLFILE = ''                     ! Location for solution data output
  WTMAX = 102                      ! Maximum weight allowed
 end-parameters

 declarations
  ITEMS: set of string             ! Index set for items
  VALUE: array(ITEMS) of real      ! Value of items
  WEIGHT: array(ITEMS) of real     ! Weight of items
  soltake: array(ITEMS) of real    ! Solution values
 end-declarations

 initializations from DATAFILE
  [VALUE,WEIGHT] as "DATA"
 end-initializations

...

 initializations to SOLFILE
  soltake as "SOL"
 end-initializations

end-model
```

13.4.4 Scalars

Besides arrays one might also wish to simply exchange scalars between the calling application and a Mosel model. One way of passing the value of a scalar to a model is to define it as a model parameter and pass the new value as an execution parameter of the model (as shown in Section 13.2). Alternatively, we might read or write scalar values in `initializations` blocks similarly to what we have seen in the previous section for arrays.

Consider the following C program: there are three scalars, `wmax`, `numitem`, and `objval`. The value of the first should be read in by the Mosel model and the last two receive solution values from the optimization run in the model.

```
#include <stdio.h>
#include "xprm_mc.h"

int wmax=100;
int numitem;
double objval;

int main()
{
 XPRMmodel mod;
 int result;
 char wmax_name[40];              /* File name of input data 'wmax' */
 char num_name[40];               /* File name of output data 'num' */
 char sol_name[40];               /* File name of solution value */
```

```
 char params[160];               /* Parameter string for model execution */

 if(XPRMinit()) return 1;        /* Initialize Mosel */

/* Prepare file names for 'initializations' using the 'raw' driver */
 sprintf(wmax_name, "mem:%p/%d", &wmax, (int)sizeof(wmax));
 sprintf(num_name, "mem:%p/%d", &numitem, (int)sizeof(numitem));
 sprintf(solution_name, "mem:%p/%d", &objval, (int)sizeof(objval));

                                 /* Pass file names as execution param.s */
 sprintf(params, "WMAX='%s',NUM='%s',SOLVAL='%s'", wmax_name, num_name,
         sol_name);

 if(XPRMexecmod(NULL, "burglar12.mos", params, &result, &mod))
  return 2;                      /* Execute a model file */

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  return 3;                      /* Test whether a solution is found */

/* Display solution values obtained from the model */
 printf("Objective value: %g\n", objval);
 printf("Total number of items: %d\n", numitem);

 XPRMresetmod(mod);

 return 0;
}
```

The Mosel model takes as execution parameters the filenames (location in memory) of the three scalars. The value `WTMAX` is initialized from the data in the application and the two other locations are written to in the `initializations to` block at the end of the model.

```
model Burglar12
 uses "mmxprs"

 parameters
  NUM = ''                       ! Location for no. of items output
  SOLVAL = ''                    ! Location for objective value output
  WMAX = ''                      ! Maximum weight allowed
 end-parameters

 declarations
  WTMAX: integer                 ! Maximum weight allowed
  ITEMS = {"camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"}     ! Index set for items
  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items
  soltake: array(ITEMS) of real  ! Solution values
 end-declarations

 VALUE :: (["camera", "necklace", "vase", "picture", "tv", "video",
            "chest", "brick"])[15,100,90,60,40,15,10,1]
 WEIGHT:: (["camera", "necklace", "vase", "picture", "tv", "video",
            "chest", "brick"])[2,20,20,30,40,30,60,10]

 initializations from 'raw:'
  WTMAX as WMAX
 end-initializations

 declarations
  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX
```

```
! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                  ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:")
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))

! Output solution to calling application
 initializations to 'raw:'
  evaluation of getobjval as SOLVAL
  evaluation of round(sum(i in ITEMS) getsol(take(i))) as NUM
 end-initializations

end-model
```

13.5 Redirecting the Mosel output

When integrating a Mosel model into an application it may be desirable to be able to redirect any output produced by the model to the application. This can be done by the means of a *callback function*. This function takes a predefined signature as shown in the following C program. If it is called from outside of the execution of any Mosel model, its parameter `model` will be `NULL`. In our example the callback function prefixes the printout of every line of Mosel output with `Mosel:`.

```
#include <stdio.h>
#include "xprm_mc.h"

/**** Callback function to handle output ****/
long XPRM_RTC cbmsg(XPRMmodel model, void *info, char *buf, unsigned long size)
{
 printf("Mosel: %.*s", (int)size, buf);
 return 0;
}

int main()
{
 int result;
 char outfile_name[40];           /* File name of output stream */

 if(XPRMinit())                   /* Initialize Mosel */
  return 1;
                                  /* Prepare file name for output stream */
                                  /* using 'cb' driver                   */
 sprintf(outfile_name, "cb:%p", cbmsg);

                                  /* Set default output stream to callback */
 XPRMsetdefstream(NULL, XPRM_F_WRITE, outfile_name);

                                  /* Execute = compile/load/run a model */
 if(XPRMexecmod(NULL, "burglar2.mos", NULL, &result, NULL))
  return 2;

 return 0;
}
```

The same procedure that has been presented here for redirecting the Mosel output can also be applied to redirect any error messages produced by Mosel—the only required modification consists in replacing the constant `XPRM_F_WRITE` by `XPRM_F_ERROR` in the argument of function `XPRMsetdefstream`.

13.6 Problem solving in C with Xpress Optimizer

In certain cases, for instance if the user wants to re-use parts of algorithms that he has written in C for the Xpress Optimizer, it may be necessary to pass from a problem formulation with Mosel to solving the problem in C by direct calls to the Optimizer. The following example shows how this may be done for the Burglar problem. We use a slightly modified version of the original Mosel model:

```
model Burglar4
 uses "mmxprs"

 declarations
  WTMAX=102                          ! Maximum weight allowed
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}           ! Index set for items

  VALUE: array(ITEMS) of real        ! Value of items
  WEIGHT: array(ITEMS) of real       ! Weight of items

  take: array(ITEMS) of mpvar        ! 1 if we take item i; 0 otherwise
 end-declarations

 initializations from 'burglar.dat'
  VALUE  WEIGHT
 end-initializations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 setparam("XPRS_LOADNAMES", true)    ! Enable loading of object names
 loadprob(MaxVal)                    ! Load problem into the optimizer

end-model
```

The procedure `maximize` to solve the problem has been replaced by `loadprob`. This procedure loads the problem into the optimizer without solving it. We also enable the loading of names from Mosel into the optimizer so that we may obtain an easily readable output.

The following C program `ugxprs1.c` reads in the Mosel model and solves the problem by direct calls to Xpress Optimizer. To be able to address the problem loaded into the optimizer, we need to retrieve the optimizer problem pointer from Mosel. Since this information is a parameter (`XPRS_PROBLEM`) that is provided by module `mmxprs`, we first need to obtain the reference of this library (by using function `XPRMfinddso`).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xprm_rt.h"
#include "xprs.h"

int main()
{
 XPRMmodel mod;
 XPRMdsolib dso;
 XPRMalltypes rvalue;
 XPRSprob prob;
 int result, ncol, len, i;
 double *sol, val;
 char *names, *onecol;
```

```
 if(XPRMinit())                          /* Initialize Mosel */
 return 1;

if((mod=XPRMloadmod("burglar4.bim", NULL))==NULL)  /* Load a BIM file */
 return 2;

if(XPRMrunmod(mod, &result, NULL))     /* Run the model (no optimization) */
 return 3;

 /* Retrieve the pointer to the problem loaded in the Optimizer */
if((dso=XPRMfinddso("mmxprs"))==NULL)
 return 4;
if(XPRMgetdsoparam(mod, dso, "xprs_problem", &result, &rvalue))
 return 5;
prob=(XPRSprob)strtoul(rvalue.ref,NULL,0);

XPRSchgobjsense(prob, XPRS_OBJ_MAXIMIZE);  /* Set sense to maximization */
if(XPRSmipoptimize(prob, ""))              /* Solve the problem */
 return 6;

if(XPRSgetintattrib(prob, XPRS_MIPSTATUS, &result))
 return 7;
                                      /* Test whether a solution is found */
if((result==XPRS_MIP_SOLUTION) || (result==XPRS_MIP_OPTIMAL))
{
 if(XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &val))
  return 8;
 printf("Objective value: %g\n", val); /* Print the objective function value */

 if(XPRSgetintattrib(prob, XPRS_INPUTCOLS, &ncol))
  return 9;
 if((sol = (double *)malloc(ncol * sizeof(double)))==NULL)
  return 10;
 if(XPRSgetsolution(prob, NULL, sol, 0, ncol-1))
  return 11;                            /* Get the primal solution values */
 if(XPRSgetnamelist(prob, 2, NULL, 0, &len, 0, ncol-1))
  return 11;                            /* Get the name array length */
 if((names = (char *)malloc(len*sizeof(char)))==NULL)
  return 12;
 if(XPRSgetnamelist(prob, 2, names, len, NULL, 0, ncol-1))
  return 13;                            /* Get the variable names */
 onecol = names;
 for(i=0; i<ncol; i++) {                /* Print out the solution */
   printf("%s: %g\n", onecol, sol[i]);
   onecol = onecol+strlen(onecol)+1;
 }
 free(names);
 free(sol);
}
return 0;
}
```

Since the Mosel language provides ample programming facilities, in most applications there will be no need to switch from the Mosel language to problem solving in C. If nevertheless this type of implementation is chosen, it should be noted that it is not possible to get back to Mosel, once the Xpress Optimizer has been called directly from C: the solution information and any possible changes made to the problem directly in the optimizer are not communicated to Mosel.

CHAPTER 14

Other programming language interfaces

In this chapter we show how the examples from Chapter 13 may be written with other programming languages, namely Java, .NET, Python, and VBA.

14.1 Java

To use the Mosel Java classes the line `import com.dashoptimization.*;` must be added at the beginning of the program.

14.1.1 Compiling and executing a model in Java

With Java Mosel is initialized by creating a new instance of class `XPRM`. To execute a Mosel model in Java we call the three Mosel functions performing the standard compile/load/run sequence as shown in the following example (file `ugcomp.java`).

```
import com.dashoptimization.*;

public class ugcomp
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;

  mosel = new XPRM();                        // Initialize Mosel

  System.out.println("Compiling  `burglar2'");
  mosel.compile("burglar2.mos");

  System.out.println("Loading  `burglar2'");
  mod = mosel.loadModel("burglar2.bim");

  System.out.println("Executing  `burglar2'");
  mod.run();

  System.out.println(" `burglar2' returned: " + mod.getResult());
 }
}
```

14.1.2 Termination

If the model execution is embedded in a larger appplication it may be useful to reset the model after its execution to free some resources allocated to it:

```
  mod.reset();                           // Reset the model
```

This will release all intermediate objects created during the execution without deleting the model itself.

It is also possible to explicitly remove the temporary directory/files created by the execution of Mosel:

```
  mosel.removeTmpDir();               // Delete temporary files
```

Unloading models or Mosel from memory can be done using the `close` method: both classes `XPRM` and `XPRMModel` implement the `AutoCloseable` interface. Terminating Mosel also removes the temporary directory/files created by the execution of Mosel.

```
  mod.close();                      // Unload a model
  mod = null;
  mosel.close();                    // Terminate Mosel
  mosel = null;
```

14.1.3 Parameters

When executing a Mosel model in Java, it is possible to pass new values for its parameters into the model. If, for instance, we want to run model 'Prime' from Section 8.3 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model), we may write the following program:

```
import com.dashoptimization.*;

public class ugparam
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  int LIM=500;

  mosel = new XPRM();                       // Initialize Mosel

  System.out.println("Compiling  `prime'");
  mosel.compile("prime.mos");

  System.out.println("Loading  `prime'");
  mod = mosel.loadModel("prime.bim");

  System.out.println("Executing  `prime'");
  mod.execParams = "LIMIT=" + LIM;
  mod.run();

  System.out.println(" `prime' returned: " + mod.getResult());
 }
}
```

Using the Mosel Java interface, it is not only possible to compile and run models, but also to access information on the different modeling objects as is shown in the following sections.

14.1.4 Accessing sets

A complete version of a program (file `ugparam.java`) for running the model 'Prime' may look as follows (we work with a model `prime2` that corresponds to the one printed in Section 8.3 but with all output printing removed because we are doing this in Java, and all entities accessed from Java are explicitly declared as `public`):

```
import com.dashoptimization.*;

public class ugparam
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
```

```
  XPRMModel mod;
  XPRMSet set;
  int LIM=500, first, last;

  mosel = new XPRM();                   // Initialize Mosel

  System.out.println("Compiling  `prime2'");
  mosel.compile("prime2.mos");

  System.out.println("Loading  `prime2'");
  mod = mosel.loadModel("prime2.bim");

  System.out.println("Executing  `prime2'");
  mod.execParams = "LIMIT=" + LIM;
  mod.run();

  System.out.println(" `prime2' returned: " + mod.getResult());

  set=(XPRMSet)mod.findIdentifier("SPrime");  // Get the object 'SPrime'
                                              // it must be a set
  if(!set.isEmpty())
  {
   first = set.getFirstIndex();         // Get the number of the first index
   last = set.getLastIndex();           // Get the number of the last index
   System.out.println("Prime numbers from 2 to " + LIM);
   for(int i=first;i<=last;i++)         // Print all set elements
    System.out.print(" " + set.getAsInteger(i) + ",");
   System.out.println();
  }
 }
}
```

To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), we retrieve the Mosel object of this name using method `findIdentifier`. If this set is not empty, then we enumerate the elements of the set (using methods `getFirstIndex` and `getLastIndex` to obtain the index range).

14.1.5 Retrieving solution values

The following program `ugsol.java` executes the model 'Burglar3' (the same as model 'Burglar2' from Chapter 2 but with all output printing removed and all model entities declared as `public`) and prints out its solution.

```
import com.dashoptimization.*;

public class ugsol
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  XPRMArray varr, darr;
  XPRMMPVar x;
  XPRMSet set;
  int[] indices;
  double val;

  mosel = new XPRM();                 // Initialize Mosel

  mosel.compile("burglar3.mos");      // Compile, load & run the model
  mod = mosel.loadModel("burglar3.bim");
  mod.run();

  if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
   System.exit(1);                    // Stop if no solution found
```

```
  System.out.println("Objective value: " + mod.getObjectiveValue());
                                      // Print the objective function value

  varr=(XPRMArray)mod.findIdentifier("take");  // Get model object 'take',
                                               // it must be an array
  darr=(XPRMArray)mod.findIdentifier("VALUE"); // Get model object 'VALUE',
                                               // it must be an array
  set=(XPRMSet)mod.findIdentifier("ITEMS");    // Get model object 'ITEMS',
                                               // it must be a set

  indices = varr.getFirstIndex();     // Get the first entry of array varr
                                      // (we know that the array is dense)
  do
  {
   x = varr.get(indices).asMPVar();   // Get a variable from varr
   val = darr.getAsReal(indices);     // Get the corresponding value
   System.out.println("take(" + set.get(indices[0]) + "): " +
                      x.getSolution() + "\t (item value: " + val + ")");
                                      // Print the solution value
  } while(varr.nextIndex(indices));   // Get the next index

  mod.close();                        // Unload the model
 }
}
```

The array of variables `varr` is enumerated using the array functions `getFirstIndex` and `nextIndex`. These methods may be applied to arrays of any type and dimension. With these functions we run systematically through all possible combinations of index tuples, hence the hint at *dense* arrays in the example. In the case of sparse arrays it is preferrable to use different enumeration functions that only enumerate those entries that are defined (see next section).

14.1.6 Sparse arrays

We now again work with the problem 'Transport' that has been introduced in Chapter 3. The objective of this problem is to calculate the flows $\mathrm{flow}_{\mathrm{pr}}$ from a set of plants to a set of sales regions that satisfy all demand and supply constraints and minimize the total cost. Not all plants may deliver goods to all regions. The flow variables $\mathrm{flow}_{\mathrm{pr}}$ are therefore defined as a *sparse* array. The following example `ugarray.java` prints out all existing entries of the array of variables.

```
import com.dashoptimization.*;

public class ugarray
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  XPRMArray varr;
  XPRMSet[] sets;
  int[] indices;
  int dim;

  mosel = new XPRM();                  // Initialize Mosel

  mosel.compile("transport.mos");      // Compile, load & run the model
  mod = mosel.loadModel("transport.bim");
  mod.run();

  varr=(XPRMArray)mod.findIdentifier("flow"); // Get model object 'flow'
                                      // it must be an array
  dim = varr.getDimension();          // Get the number of dimensions
                                      // of the array
  sets = varr.getIndexSets();         // Get the indexing sets
```

```
  indices = varr.getFirstTEIndex();    // Get the first true entry index
  do
  {
   System.out.print("flow(");
   for(int i=0;i<dim-1;i++)
    System.out.print(sets[i].get(indices[i]) + ",");
   System.out.print(sets[dim-1].get(indices[dim-1]) + "), ");
  } while(varr.nextTEIndex(indices));  // Get next true entry index tuple
  System.out.println();

  mod.close();                         // Unload the model
 }
}
```

In this example, we first get the number of indices (dimensions) of the array of variables `varr` (using method `getDimension`). We use this information to enumerate the entries of every index tuple for generating a nicely formatted output. The array `sets` holds all the index sets of `varr` and the array `indices` corresponds to a single index tuple.

The enumeration starts with the first defined index tuple, obtained with method `getFirstTEIndex`, and continues with a series of calls to `nextTEIndex` until all defined entries have been enumerated.

14.1.7 Exchanging data between an application and a model

In the previous examples we have seen how to retrieve information about the model objects from a Mosel model after its execution. In all cases the input data is defined in the model itself or read in from an external (text) file. However, when embedding a model into an application frequently the input data for the model will be stored (or generated by) the application itself. In such a case the user will certainly wish a more immediate means of communication to the model than having to write the input data to an external text file or database. In the following two subsections we therefore show how to pass data in memory from an application to a Mosel model, and with the same mechanism (namely, using the `jraw` I/O driver) from the model back to the calling application.

14.1.7.1 Dense arrays

As a first example we shall look at the case of *dense arrays* holding the input and solution data. In the underlying Mosel model this corresponds to arrays indexed by range sets that are known in the model before the data are read in. In this example, we shall work with a version of the 'Burglar model based on the very first version we have seen in Section 2.1 where all arrays are indexed by the range set `ITEMS = 1..8`.

The following Java program `ugiodense.java` compiles, loads, and runs a Mosel model and then prints out the solution values. The input data (arrays `vdata` and `wdata`) and the array `solution` that is to receive the solution values are passed on to the model through model *parameters*. Communication of the data between the application and the Mosel model is achieved through the `jraw` I/O driver. File names for this driver have the form *`jrawoption[,...],filename`*, where *`filename`* is an object reference. Since we are working with dense, one-dimensional arrays we use the option `noindex`, indicating that only the data and not the index tuples are to be exchanged.

```
import com.dashoptimization.*;

public class ugiodense
{                                      // Input data
 static final double[] vdata={15,100,90,60,40,15,10, 1};   // VALUE
 static final double[] wdata={ 2, 20,20,30,40,30,60,10};   // WEIGHT

                                       // Array to receive solution values
 static double[] solution = new double[8];
```

```
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;

  mosel = new XPRM();                 // Initialize Mosel

  mosel.compile("burglar8.mos");      // Compile & load the model
  mod = mosel.loadModel("burglar8.bim");

                      // Associate the Java objects with names in Mosel
  mosel.bind("vdat", vdata);
  mosel.bind("wdat", wdata);
  mosel.bind("sol", solution);
                      // File names are passed through execution parameters
  mod.execParams =
   "VDATA='noindex,vdat',WDATA='noindex,wdat',SOL='noindex,sol'";

  mod.run();                          // Run the model

  if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
   System.exit(1);                    // Stop if no solution found

                      // Display solution values obtained from the model
  System.out.println("Objective value: " + mod.getObjectiveValue());
  for(int i=0;i<8;i++)
   System.out.println(" take(" + (i+1) + "): " + solution[i]);

  mod.close();                        // Unload the model
 }
}
```

The model file `burglar8.mos` is the same as model `burglar6.mos` from Section 13.4.1 with the only difference that the name of the I/O driver in the `initializations` blocks now is `jraw` instead of `raw`, such as:

```
 initializations from 'jraw:'
  VALUE as VDATA  WEIGHT as WDATA
 end-initializations
```

14.1.7.2 Sparse arrays

Let us now study the probably more frequent case of data stored in *sparse format*. In the Mosel model (`burglar9.mos`) we use a set of strings instead of a simple range set to index the various arrays and in the Java program (`ugiosparse.java`) we need to define slightly more complicated structures to hold the indices and the data entries. To save us writing out the indices twice, we have grouped the two input data arrays into a single class. When passing the data arrays to the Mosel model we now do not use any option, meaning that data is transferred in sparse format. Instead, we now need to indicate which fields of the Java objects are to be selected (in brackets after the object reference).

```
import com.dashoptimization.*;

public class ugiosparse
{
                        // Class to store initial values for array 'data'
 public static class MyData
 {
  public String ind;                // index name
  public double val,wght;           // value and weight data entries
  MyData(String i, double v, double w)
  { ind=i; val=v; wght=w; }
 }
                        // Class to receive solution values
 public static class MySol
 {
```

```
  public String ind;                // index name
  public double val;                // solution value
 }

 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  MyData data[]={new MyData("camera",15,2), new MyData("necklace",100,20),
                 new MyData("vase",90,20), new MyData("picture",60,30),
   new MyData("tv",40,40), new MyData("video",15,30),
                 new MyData("chest",10,60), new MyData("brick",1,10)};
  MySol[] solution=new MySol[8];

  for(int i=0;i<8;i++) solution[i] = new MySol();

  mosel = new XPRM();                // Initialize Mosel

  mosel.compile("burglar9.mos");     // Compile & load the model
  mod = mosel.loadModel("burglar9.bim");

                        // Associate the Java objects with names in Mosel
  mosel.bind("dt", data);
  mosel.bind("sol", solution);
                        // File names are passed through execution parameters
  mod.execParams = "DATA='dt(ind,val,wght)',SOL='sol(ind,val)'";

  mod.run();                         // Run the model

  if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
   System.exit(1);                   // Stop if no solution found

                        // Display solution values obtained from the model
  System.out.println("Objective value: " + mod.getObjectiveValue());
  for(int i=0;i<8;i++)
   System.out.println(" take(" + solution[i].ind + "): " + solution[i].val);

  mod.close();                       // Unload the model
 }
}
```

The model `burglar9.mos` run by this program is the same as the model `burglar7.mos` displayed in Section 13.4.2, but using the I/O driver `jraw` instead of `raw`.

14.1.7.3 Dynamic data

The two examples of in-memory communication of dense and sparse data in the preceding sections have in commun that all data structures in the application, and in particular the structures to receive output data, are of fixed size. We therefore now introduce an alternative communication mechanism working with streams, that enables dynamic sizing of data structures on the application level, a feature that is particularly useful for solution output where effective data sizes are not known a priori. This communication mechanism employs the I/O driver `java` (see also Section 14.1.8). The main part of our Java program (file `ugiocb.java`) now looks as follows.

```
public static modelInit cbinit=new modelInit();

public static void main(String[] args) throws Exception
{
 XPRM mosel;
 XPRMModel mod;

 mosel = new XPRM();                // Initialize Mosel

 mosel.compile("burglar13.mos");    // Compile & load the model
 mod = mosel.loadModel("burglar13.bim");
```

```
                      // File names are passed through execution parameters
 mod.execParams = "DATAFILE='java:ugiocb.cbinit'," +
                  "SOLFILE='java:ugiocb.cbinit'";

 mod.run();                           // Run the model

 if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
  System.exit(1);                     // Stop if no solution found

                      // Display solution values obtained from the model
 System.out.println("Objective value: " + mod.getObjectiveValue());
 for(int i=0;i<solsize;i++)
  System.out.println(" take(" + solution[i].ind + "): " + solution[i].val);

 mod.close();                         // Unload the model
}
```

The information passed to the model in the runtime parameters now is an instance of a class that implements interfaces for initialization from and to streams as shown below. The functionality for *dynamic output retrieval* employs the Mosel library functions that we have already seen in Sections 14.1.4 and 14.1.5 for accessing models after their termination. The *dynamic data input* to a Mosel model uses a new set of dedicated functions that are explained with some more detail after the program extract.

```
static final double[] vdata={15,100,90,60,40,15,10, 1};   // VALUE
static final double[] wdata={ 2, 20,20,30,40,30,60,10};   // WEIGHT
static final String[] ind={"camera", "necklace", "vase", "picture",
    "tv", "video", "chest", "brick"};                     // Index names
static final int datasize=8;

public static class MySol {
 public String ind;                // index name
 public double val;                // solution value
}
static MySol[] solution;
static int solsize;


public static class modelInit implements XPRMInitializationFrom, XPRMInitializationTo
{
 public boolean initializeTo(String label, XPRMValue value)
 {
  XPRMArray solarr;
  XPRMSet[] sets;
  int[] indices;
  int ct;

  if(label.equals("SOL"))
  {
   solarr=(XPRMArray)value;
   solsize=solarr.getSize();
   solution = new MySol[solsize];
   for(int i=0;i<solsize;i++) solution[i] = new MySol();

   sets = solarr.getIndexSets();          // Get the indexing sets
   ct=0;
   indices = solarr.getFirstTEIndex();    // Get the first entry of the array
   do
   {
    solution[ct].ind=sets[0].getAsString(indices[0]);
    solution[ct].val=solarr.getAsReal(indices);
    ct++;
   } while(solarr.nextTEIndex(indices));  // Get the next index
  }
  else System.out.println("Unknown output data item: " + label + "=" + value);
  return true;
 }
```

```
 public boolean initializeFrom(XPRMInitializeContext ictx, String label, XPRMTyped type)
 {
  try
  {
   if(label.equals("DATA"))
   {
    ictx.sendControl(ictx.CONTROL_OPENLST);
    for(int i=0;i<datasize;i++)
    {
     ictx.sendControl(ictx.CONTROL_OPENNDX);
      ictx.send(ind[i]);
     ictx.sendControl(ictx.CONTROL_CLOSENDX);
     ictx.sendControl(ictx.CONTROL_OPENLST);
      ictx.send(vdata[i]);
      ictx.send(wdata[i]);
     ictx.sendControl(ictx.CONTROL_CLOSELST);
    }
    ictx.sendControl(ictx.CONTROL_CLOSELST);
    return true;
   }
   else
   {
    System.err.println("Label  `"+label+"' not found.");
    return false;
   }
  }
  catch(java.io.IOException e)
  {
   System.err.println(" `"+label+"' could not be initialized - "+e);
   return false;
  }
 }
}
```

The format used to represent data for *dynamic data input* is the same as the default text format used by `initializations` blocks. For example, the array definition

```
mydata: [ ("ind1" 3) [5 1.2] ("ind2" 7) [4 6.5] ]
```

is represented by the following sequence of function calls:

```
ictx.sendControl(ictx.CONTROL_OPENLST);     ! [
ictx.sendControl(ictx.CONTROL_OPENNDX);     !   (
ictx.send("ind1");                          !     "ind1"
ictx.send(3);                               !     3
ictx.sendControl(ictx.CONTROL_CLOSENDX);    !   )
ictx.sendControl(ictx.CONTROL_OPENLST);     !   [
ictx.send(5);                               !     5
ictx.send(1.2);                             !     1.2
ictx.sendControl(ictx.CONTROL_CLOSELST);    !   ]
ictx.sendControl(ictx.CONTROL_OPENNDX);     !   (
ictx.send("ind2");                          !     "ind2"
ictx.send(7);                               !      7
ictx.sendControl(ictx.CONTROL_CLOSENDX);    !   )
ictx.sendControl(ictx.CONTROL_OPENLST);     !   [
ictx.send(4);                               !     4
ictx.send(6.5);                             !     6.5
ictx.sendControl(ictx.CONTROL_CLOSELST);    !   ]
ictx.sendControl(ictx.CONTROL_CLOSELST);    ! ]
```

The `send` and `sendControl` methods may take an additional last argument indicating whether data is to be processed immediately or only once the queue of tokens is full (default).

With Java, we use exactly the same model file `burglar13.mos` as with C (see Section 13.4.3 for the listing).

14.1.7.4 Scalars

Besides arrays one might also wish to simply exchange scalars between the calling application and a Mosel model. One way of passing the value of a scalar to a model is to define it as a model parameter and pass the new value as an execution parameter to the model (as shown in Section 14.1.3). Alternatively, we might read or write scalar values in `initializations` blocks similarly to what we have seen in the previous section for arrays.

Consider the following Java program: we wish to exchange the values of the three scalars, `wmax`, `numitem`, and `objval` with the Mosel model run by this program. The value of the first scalar should be read in by the Mosel model and the last two receive solution values from the optimization run in the model. Since it is not possible to address scalars directly from the model we have collected them into a class `MyData` the fields of which are then specified in the execution parameters as the locations of the data.

```
import com.dashoptimization.*;

public class ugioscalar
{
 public static class MyData            // Scalars for data in/output
 {
  public int wmax;
  public int numitem;
  public double objval;
 }

 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  MyData data=new MyData();

  data.wmax=100;

  mosel = new XPRM();                 // Initialize Mosel

  mosel.compile("burglar11.mos");     // Compile & load the model
  mod = mosel.loadModel("burglar11.bim");

                     // Associate the Java object with a name in Mosel
  mosel.bind("data", data);
                     // File names are passed through execution parameters
  mod.execParams =
   "WMAX='data(wmax)',NUM='data(numitem)',SOLVAL='data(objval)'";

  mod.run();                          // Run the model

  if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
   System.exit(1);                    // Stop if no solution found

                     // Display solution values obtained from the model
  System.out.println("Objective value: " + data.objval);
  System.out.println("Total number of items: " + data.numitem);

  mod.close();                        // Unload the model
 }
}
```

The Mosel model `burglar11.mos` run by this program is the same as the model `burglar12.mos` displayed in Section 13.4.4, but using the I/O driver `jraw` instead of `raw`. This model takes as execution parameters the filenames (location in memory) of the three scalars. The integer `WTMAX` is initialized from the value in the Java application and the two other locations are written to in the `initializations to` block at the end of the model.

14.1.8 Redirecting the Mosel output

When executing a Mosel model from a Java application it may be desirable to be able to process the output produced by Mosel directly in the application. The following Java program `ugcb.java` shows a callback-style functionality that redirects the Mosel standard output to an `OutputStream` object which is used to prefix every line of Mosel output with the string `Mosel:` before printing it.

To redirect Mosel streams to a Java object (Java streams or `ByteBuffer`) we need to use the `java` I/O driver. The same mechanism that is used here for redirecting the output stream of Mosel (indicated by `XPRM.F_OUTPUT`, with the additional option `XPRM.F_LINBUF` to enable line buffering) can equally be used to redirect, for instance, the error stream (denoted by the constant `XPRM.F_ERROR`).

```
import java.io.*;
import com.dashoptimization.*;

public class ugcb
{
                            // OutputStream class to handle default output
 public static class MyOut extends OutputStream
 {
  public void flush()
  { System.out.flush(); }
  public void write(byte[] b)
  {
   System.out.print("Mosel: ");
   System.out.write(b,0,b.length);
  }
  // These methods are not used by Mosel:
  public void write(byte[] b,int off,int len) {}
  public void write(int b) {}
  public void close() {}
 }

 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  MyOut cbmsg = new MyOut();            // Define output stream as "MyOut"

  mosel = new XPRM();                   // Initialize Mosel

  mosel.bind("mycb", cbmsg);    // Associate Java object with a name in Mosel
                                // Set default output stream to cbmsg
  mosel.setDefaultStream(XPRM.F_OUTPUT|XPRM.F_LINBUF, "java:mycb");

  mosel.compile("burglar2.mos");        // Compile, load & run the model
  mod = mosel.loadModel("burglar2.bim");
  mod.run();
 }
}
```

14.2 .NET

Example code in this guide will be in C#, however one can access the Mosel .NET interface via other languages that target the .NET Framework, such as VB.NET.

To use the Mosel .NET classes the line `using Mosel;` must be added at the beginning of the program, and your project should have a dependency on the Xpress package `FICO.Xpress.XPRMdn`. This package is available in the `lib/nuget` folder of the Xpress installation. To make the packages visible to NuGet, this folder should be configured as a NuGet local package source.

14.2.1 Compiling and executing a model in C#

With C# Mosel is initialized by obtaining a new instance of class `XPRM` via the static method `XPRM.Init()`. To execute a Mosel model in C# we call the three Mosel functions performing the standard compile/load/run sequence as shown in the following example (file `ugcomptmp.cs`).

```
using System;
using System.IO;
using Mosel;


namespace ugcomptmp.cs {

  public class ugcomptmp {
    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile the Mosel model, save the BIM file in Mosel's temp. dir.
      mosel.Compile("", "burglar2.mos", "tmp:burglar2.bim");

      // Load the BIM file
      XPRMModel model = mosel.LoadModel("tmp:burglar2.bim");

      // Run the model
      model.Run();
    }
  }

}
```

14.2.2 Termination

If the model execution is embedded in a larger appplication it may be useful to reset the model after its execution to free some resources allocated to it:

```
  mod.Reset();                          // Reset the model
```

This will release all intermediate objects created during the execution without deleting the model itself.

It is also possible to explicitly remove the temporary directory/files created by the execution of Mosel:

```
  mosel.RemoveTmpDir();                 // Delete temporary files
```

Unloading models or Mosel from memory is ensured through standard disposal, finalization + garbage collection functionalities of the .NET runtime. The disposal methods are public and may be called from the user's C# program. Finalization of Mosel only takes effect once all loaded models have been finalized or disposed. Finalizing or diposing Mosel also removes the temporary directory/files created by the execution of Mosel.

```
  mod.Dispose();                        // Dispose a model
  mod = null;
  mosel.Dispose();                      // Dispose Mosel
  mosel = null;
```

14.2.3 Parameters

When executing a Mosel model in C#, it is possible to pass new values for its parameters into the model. If, for instance, we want to run model 'Prime' from Section 8.3 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model), we may write the following

program:

```
using System;
using System.IO;
using Mosel;


namespace ugparam.cs {

  public class ugparam {
    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      XPRMSet set;
      int LIM=500, first, last;

      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load a model
      XPRMModel model = mosel.CompileAndLoad("prime.mos");

      // Run the model
      model.ExecParams = "LIMIT=" + LIM;
      model.Run();
      Console.WriteLine(" `prime' returned: " + model.Result);
    }
  }
}
```

Using the Mosel .NET interface, it is not only possible to compile and run models, but also to access information on the different modeling objects as is shown in the following sections.

14.2.4 Accessing sets

A complete version of a program (file `ugparam.cs`) for running the model 'Prime' may look as follows (we work with a model `prime2` that corresponds to the one printed in Section 8.3 but with all output printing removed because we are doing this in C#, and all entities accessed from C# are declared as `public`):

```
using System;
using System.IO;
using Mosel;


namespace ugparam.cs {

  public class ugparam {
    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      XPRMSet set;
      int LIM=500, first, last;

      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load a model
      XPRMModel model = mosel.CompileAndLoad("prime.mos");

      // Run the model
      model.ExecParams = "LIMIT=" + LIM;
```

```
      model.Run();
      Console.WriteLine(" `prime' returned: " + model.Result);

      // Get model object 'SPrime', it must be a set
      set=(XPRMSet)model.FindIdentifier("SPrime");

      // Enumerate the set elements
      if (!set.IsEmpty)
      {
        first = set.FirstIndex;         // Get the number of the first index
        last = set.LastIndex;           // Get the number of the last index
        Console.WriteLine("Prime numbers from 2 to " + LIM);
        for (int i=first;i<=last;i++)   // Print all set elements
          Console.Write(" {0},", set.GetAsInteger(i));
        Console.WriteLine();
      }
    }
  }
}
```

To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), we retrieve the Mosel object of this name using method `FindIdentifier`. If this set is not empty, then we enumerate the elements of the set (using properties `FirstIndex` and `LastIndex` to obtain the index range).

14.2.5 Retrieving solution values

The following program `ugsol.cs` executes the model 'Burglar3' (the same as model 'Burglar2' from Chapter 2 but with all output printing removed and all model entities declared as `public`) and prints out its solution.

```
using System;
using System.IO;
using Mosel;


namespace ugsol.cs {

  public class ugsol {
    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static int Main(string[] args) {
      XPRMArray varr, darr;
      XPRMSet set;
      XPRMMPVar x;
      double val;

      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load a model
      XPRMModel model = mosel.CompileAndLoad("burglar3.mos");

      // Run the model
      model.Run();

      if(model.ProblemStatus!=XPRMProblemStatus.PB_OPTIMAL)
        return 1;                     // Stop if no solution found

      Console.WriteLine("Objective value: " + model.ObjectiveValue);
                                      // Print the objective function value

      // Get model object 'take', it must be an array
```

```
      varr=(XPRMArray)model.FindIdentifier("take");
      // Get model object 'VALUE', it must be an array
      darr=(XPRMArray)model.FindIdentifier("VALUE");
      // Get model object 'ITEMS', it must be a set
      set=(XPRMSet)model.FindIdentifier("ITEMS");

      // Enumerate all entries of 'varr' (dense array)
      foreach(int[] indices in varr.Indices)
      {
        x = varr.Get(indices).AsMPVar();   // Get a variable from varr
        val = darr.GetAsReal(indices);     // Get the corresponding value
//      Console.WriteLine("take(" + set.GetAsString(indices[0]) + "): " +
        Console.WriteLine("take" + varr.IndexToString(indices) + ": " +
                  x.Solution + "\t (item value: " + val + ")");
      }

      model.Reset();                          // Reset the model
      return 0;
    }
  }
}
```

The array of variables `varr` is enumerated using via the array `Indices` property. This may be applied to arrays of any type and dimension. With these functions we run systematically through all possible combinations of index tuples, hence the hint at *dense* arrays in the example. In the case of sparse arrays it is preferrable to use a different enumeration property that only enumerate those entries that are defined (see next section).

14.2.6 Sparse arrays

We now again work with the problem 'Transport' that has been introduced in Chapter 3. The objective of this problem is to calculate the flows $\mathrm{flow}_{\mathrm{pr}}$ from a set of plants to a set of sales regions that satisfy all demand and supply constraints and minimize the total cost. Not all plants may deliver goods to all regions. The flow variables $\mathrm{flow}_{\mathrm{pr}}$ are therefore defined as a *sparse* array. The following example `ugarray.cs` prints out all existing entries of the array of variables.

```
using System;
using System.IO;
using Mosel;


namespace ugarray.cs {

  public class ugarray {
    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      XPRMArray varr;
      XPRMSet[] sets;
      XPRMValue[] vindex;
      int dim;

      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load a model
      XPRMModel model = mosel.CompileAndLoad("transport.mos");

      // Run the model
      model.Run();

      // Get model object 'flow', it must be an array
```

```
      varr=(XPRMArray)model.FindIdentifier("flow");
      dim = varr.Dim;           // Get the number of dimensions of the array
      sets = varr.IndexSets;    // Get the indexing sets

      // Enumerate over the true entries
      foreach(int[] indices in varr.TEIndices)
      {
        // Get the values for this index
        vindex = varr.DereferenceIndex(indices);
        Console.Write("flow(");
        for(int i=0;i<dim-1;i++)
          Console.Write(vindex[i] + ",");
        Console.Write(vindex[dim-1] + "), ");

        // Alternative printing method:
        // Console.Write("flow" + varr.IndexToString(indices) + ", ");
      }
      Console.WriteLine();

      model.Reset();                        // Reset the model
    }
  }
}
```

In this example, we first get the number of indices (dimensions) of the array of variables `varr` (using property `Dim`). We use this information to enumerate the entries of every index tuple for generating a nicely formatted output. The array `sets` holds all the index sets of `varr` and the array `indices` corresponds to a single index tuple.

The enumeration runs over all the defined index tuples, obtained with property `TEIndices`.

14.2.7 Exchanging data between an application and a model

In the previous examples we have seen how to retrieve information about the model objects from a Mosel model after its execution. In all cases the input data is defined in the model itself or read in from an external (text) file. However, when embedding a model into an application frequently the input data for the model will be stored (or generated by) the application itself. In such a case the user will certainly wish a more immediate means of communication to the model than having to write the input data to an external text file or database. In the following two subsections we therefore show how to pass data in memory from an application to a Mosel model, and with the same mechanism (namely, using the `dotnetraw` I/O driver) from the model back to the calling application.

14.2.7.1 Dense arrays

As a first example we shall look at the case of *dense arrays* holding the input and solution data. In the underlying Mosel model this corresponds to arrays indexed by range sets that are known in the model before the data are read in. In this example, we shall work with a version of the Burglar model based on the very first version we have seen in Section 2.1 where all arrays are indexed by the range set `ITEMS = 1..8`.

The following C# program `ugiodense.cs` compiles, loads, and runs a Mosel model and then prints out the solution values. The input data (arrays `vdata` and `wdata`) and the array `solution` that is to receive the solution values are passed on to the model through model *parameters*. Communication of the data between the application and the Mosel model is achieved through the `dotnetraw` I/O driver. File names for this driver have the form *`dotnetrawoption[,...],filename`*, where *`filename`* is an object reference. Since we are working with dense, one-dimensional arrays we use the option `noindex`, indicating that only the data and not the index tuples are to be exchanged.

```
using System;
using System.IO;
using Mosel;


namespace ugiodense.cs {

  public class ugiodense {

    /// <summary>
    /// Arrays containing initialization data for the model
    /// </summary>
    static double[] vdata = new double[] {15,100,90,60,40,15,10, 1};  // VALUE
    static double[] wdata = new double[] { 2, 20,20,30,40,30,60,10};  // WEIGHT

    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load the Mosel model
      XPRMModel model = mosel.CompileAndLoad("burglar8d.mos");

      // Associate the .NET objects with names in Mosel
      model.Bind("vdat", vdata);
      model.Bind("wdat", wdata);

      // Create a new array for solution data and bind that to the name 'SOL'
      double[] solution = new double[8];
      mosel.Bind("sol", solution);

      // Pass data location as a parameter to the model
      model.ExecParams = "VDATA='noindex,vdat',WDATA='noindex,wdat',SOL='noindex,sol'";

      // Run the model
      model.Run();

      // Print the solution
      Console.WriteLine("Objective value: {0}", model.ObjectiveValue);
      for (int i=0;i<8;i++)
        Console.Write(" take({0}): {1}", (i+1), solution[i]);
      Console.WriteLine();
    }
  }

}
```

The model file `burglar8d.mos` is the same as model `burglar6.mos` from Section 13.4.1 with the only difference that the name of the I/O driver in the `initializations` blocks now is `dotnetraw` instead of `raw`, such as:

```
initializations from 'dotnetraw:'
 VALUE as VDATA  WEIGHT as WDATA
end-initializations
```

14.2.7.2 Sparse arrays

Let us now study the probably more frequent case of data stored in *sparse format*. In the Mosel model (`burglar9d.mos`) we use a set of strings instead of a simple range set to index the various arrays and in the C# program (`ugiosparse.cs`) we need to define slightly more complicated structures to hold the indices and the data entries. To save us writing out the indices twice, we have grouped the two input data arrays into a single class. When passing the data arrays to the Mosel model we now do not use any

option, meaning that data is transferred in sparse format. Instead, we now need to indicate which fields of the C# objects are to be selected (in brackets after the object reference).

```
using System;
using System.IO;
using Mosel;


namespace ugiosparse.cs {

  public class ugiosparse {

    /// <summary>
    /// Arrays containing initialization data for the model
    /// </summary>
    static double[] vdata = new double[] {15,100,90,60,40,15,10, 1};  // VALUE
    static double[] wdata = new double[] { 2, 20,20,30,40,30,60,10};  // WEIGHT

    /// <summary>
    /// Structure to store initial values for the array 'data'
    /// </summary>
    class MyData {
      public string ind;
      public double val;
      public double wght;

      public MyData(string i, double v, double w) {
        this.ind = i;
        this.val = v;
        this.wght = w;
      }
    }

    /// <summary>
    /// Structure to receive solution values
    /// </summary>
    class MySol {
      public string ind;
      public double val;
    }

    /// <summary>
    /// The initial values for the array 'data'
    /// </summary>
    private static MyData[] data = new MyData[] {
                 new MyData("camera",15,2), new MyData("necklace",100,20),
                 new MyData("vase",90,20), new MyData("picture",60,30),
   new MyData("tv",40,40), new MyData("video",15,30),
                 new MyData("chest",10,60), new MyData("brick",1,10) };


    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load the Mosel model
      XPRMModel model = mosel.CompileAndLoad("burglar9d.mos");

      // Associate the .NET object with a name in Mosel
      model.Bind("dt", data);

      // Create a new array for solution data and bind that to the name 'SOL'
      MySol[] solution=new MySol[8];
      for(int i=0;i<8;i++) solution[i] = new MySol();
      mosel.Bind("sol", solution);
```

```
      // Pass data location as a parameter to the model
      model.ExecParams = "DATA='dt(ind,val,wght)',SOL='sol(ind,val)'";

      // Run the model
      model.Run();

      // Print the solution
      Console.WriteLine("Objective value: {0}", model.ObjectiveValue);
      for (int i=0;i<8;i++)
        Console.Write(" take({0}): {1}",  solution[i].ind, solution[i].val);
      Console.WriteLine();
    }
  }

}
```

The model `burglar9d.mos` run by this program is the same as the model `burglar7.mos` displayed in Section 13.4.2, but using the I/O driver `dotnetraw` instead of `raw`.

14.2.7.3 Dynamic data

The two examples of in-memory communication of dense and sparse data in the preceding sections have in commun that all data structures in the application, and in particular the structures to receive output data, are of fixed size. We therefore now introduce an alternative communication mechanism working with streams, that enables dynamic sizing of data structures on the application level, a feature that is particularly useful for solution output where effective data sizes are not known a priori. This communication mechanism employs the I/O driver `dotnet` (see also Section 14.2.8). The main part of our C# program (file `ugiocb.cs`) now looks as follows.

```
    [STAThread]
    static int Main(string[] args) {
      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Compile and load the Mosel model
      XPRMModel model = mosel.CompileAndLoad("burglar13.mos");

      // Set the execution parameters and bind the variables
      model.SetExecParam("DATAFILE","dotnet:cbinitfrom");
      model.SetExecParam("SOLFILE","dotnet:cbinitto");
      model.Bind("cbinitfrom", new XPRMInitializationFrom(initializeFrom));
      model.Bind("cbinitto", new XPRMInitializationTo(initializeTo));

      // Run the model
      model.Run();

      if(model.ProblemStatus!=XPRMProblemStatus.PB_OPTIMAL)
        return 1;                       // Stop if no solution found

     // Display solution values obtained from the model
      Console.WriteLine("Objective value: {0}", model.ObjectiveValue);
      for(int i=0;i<solsize;i++)
        Console.WriteLine(" take({0}): {1}", solution[i].ind, solution[i].val);

      model.Reset();                    // Reset the model
      return 0;
    }
```

The information passed to the model in the runtime parameters are now instances of delegates for initialization from and to streams as shown below. The functionality for *dynamic output retrieval* employs the Mosel library functions that we have already seen in Sections 14.2.4 and 14.2.5 for accessing models after their termination. The *dynamic data input* to a Mosel model uses a new set of dedicated functions that are explained with some more detail after the program extract.

```
    static double[] vdata=new double[] {15,100,90,60,40,15,10, 1};  // VALUE
    static double[] wdata=new double[] { 2, 20,20,30,40,30,60,10};  // WEIGHT
    static string[] ind=new string[] {"camera", "necklace", "vase", "picture",
                  "tv", "video", "chest", "brick"};            // Index names
    static int datasize=8;

    /// <summary>
    /// Structure to receive solution values
    /// </summary>
    class MySol {
      public string ind;                // index name
      public double val;                // solution value
    }
    static MySol[] solution;
    static int solsize;

    /// <summary>
    /// A function to initialize the Mosel data-structures via callback
    /// </summary>
    public static bool initializeFrom(XPRMInitializeContext ictx,string label,XPRMTyped type)
    {
      try {
        switch (label) {
          case "DATA":
            ictx.Send(XPRMInitializeControl.OpenList);
            for (int i=0;i<datasize;i++) {
              ictx.Send(XPRMInitializeControl.OpenIndices);
                ictx.Send(ind[i]);
              ictx.Send(XPRMInitializeControl.CloseIndices);
              ictx.Send(XPRMInitializeControl.OpenList);
                ictx.Send(vdata[i]);
                ictx.Send(wdata[i]);
              ictx.Send(XPRMInitializeControl.CloseList);
            }
            ictx.Send(XPRMInitializeControl.CloseList);
            return true;
          default:
            Console.WriteLine("Label '{0}' not found", label);
            return false;
        }
      } catch (Exception e) {
        Console.WriteLine("Label '{0}' could not be initialized - {1}", label, e.Message);
        return false;
      }
    }


    /// <summary>
    /// A method to retrieve data from Mosel
    /// </summary>
    public static bool initializeTo(string label,XPRMValue val) {
//      Console.WriteLine(".NET: {0} = {1}", label, val);

      XPRMArray solarr;
      XPRMValue[] vindex;

      switch (label) {
        case "SOL":
          solarr=(XPRMArray)val;
          solsize=solarr.Size;
          solution = new MySol[solsize];
          for(int i=0;i<solsize;i++) solution[i] = new MySol();

          int ct=0;
          // Enumerate solarr as sparse array
          foreach(int [] indices in solarr.TEIndices) {
            vindex = solarr.DereferenceIndex(indices);
            solution[ct].ind = vindex[0].AsString();
            solution[ct].val = solarr.GetAsReal(indices);
```

```
          ct++;
        }
        return true;
      default:
         Console.WriteLine("Unknown output data item: '{0}'={1} not found", label, val);
         return false;
    }
  }
```

The format used to represent data for *dynamic data input* is the same as the default text format used by `initializations` blocks. For example, the array definition

```
mydata: [ ("ind1" 3) [5 1.2] ("ind2" 7) [4 6.5] ]
```

is represented by the following sequence of function calls:

```
ictx.Send(XPRMInitializeControl.OpenList);      ! [
ictx.Send(XPRMInitializeControl.OpenIndices);   !   (
ictx.Send("ind1");                              !     "ind1"
ictx.Send(3);                                   !     3
ictx.Send(XPRMInitializeControl.CloseIndices);  !   )
ictx.Send(XPRMInitializeControl.OpenList);      !   [
ictx.Send(5);                                   !     5
ictx.Send(1.2);                                 !     1.2
ictx.Send(XPRMInitializeControl.CloseList);     !   ]
ictx.Send(XPRMInitializeControl.OpenIndices);   !   (
ictx.Send("ind2");                              !     "ind2"
ictx.Send(7);                                   !      7
ictx.Send(XPRMInitializeControl.CloseIndices);  !   )
ictx.Send(XPRMInitializeControl.OpenList);      !   [
ictx.Send(4);                                   !     4
ictx.Send(6.5);                                 !     6.5
ictx.Send(XPRMInitializeControl.CloseList);     !   ]
ictx.Send(XPRMInitializeControl.CloseList);     ! ]
```

The `send` and `sendControl` methods may take an additional last argument indicating whether data is to be processed immediately or only once the queue of tokens is full (default).

With C#, we use exactly the same model file `burglar13.mos` as with C (see Section 13.4.3 for the listing).

14.2.7.4 Scalars

Besides arrays one might also wish to simply exchange scalars between the calling application and a Mosel model. One way of passing the value of a scalar to a model is to define it as a model parameter and pass the new value as an execution parameter to the model (as shown in Section 14.2.3). Alternatively, we might read or write scalar values in `initializations` blocks similarly to what we have seen in the previous section for arrays.

Consider the following C# program: we wish to exchange the values of the three scalars, `wmax`, `numitem`, and `objval` with the Mosel model run by this program. The value of the first scalar should be read in by the Mosel model and the last two receive solution values from the optimization run in the model. Since it is not possible to address scalars directly from the model we have collected them into a class `MyData` the fields of which are then specified in the execution parameters as the locations of the data.

```
using System;
using System.IO;
using Mosel;


namespace ugioscalar.cs {
```

```
 public class ugioscalar {
   /// <summary>
   /// Structure to receive solution values
   /// </summary>
   class MyData {
     public int wmax;
     public int numitem;
     public double objval;
   }

   /// <summary>
   /// Main entry point for the application
   /// </summary>
   [STAThread]
   static int Main(string[] args) {
     MyData data=new MyData();
     data.wmax=100;

     // Initialize Mosel
     XPRM mosel = XPRM.Init();

     // Compile and load a model
     XPRMModel model = mosel.CompileAndLoad("burglar11.mos");

     // Associate the .NET object with a name in Mosel
     model.Bind("data", data);

     // Run the model, passing data location as parameters
     model.ExecParams =
       "WMAX='data(wmax)',NUM='data(numitem)',SOLVAL='data(objval)'," +
"IODRV='dotnetraw:'";
     model.Run();

     if(model.ProblemStatus!=XPRMProblemStatus.PB_OPTIMAL)
       return 1;                    // Stop if no solution found

                      // Display solution values obtained from the model
     Console.WriteLine("Objective value: " + data.objval);
     Console.WriteLine("Total number of items: " + data.numitem);

     model.Reset();                               // Reset the model
     return 0;
   }
 }
}
```

The Mosel model `burglar11.mos` run by this program is the same as the model `burglar12.mos` displayed in Section 13.4.4, but using the I/O driver `dotnetraw` instead of `raw` (which we set through the `IODRV` parameter). This model takes as execution parameters the filenames (location in memory) of the three scalars. The integer `WTMAX` is initialized from the value in the .NET application and the two other locations are written to in the `initializations to` block at the end of the model.

14.2.8 Redirecting the Mosel output

When executing a Mosel model from a .NET application it may be desirable to be able to process the output produced by Mosel directly in the application. The following C# program `ugcb.cs` shows a callback-style functionality that redirects the Mosel standard output to a `TextWriter` object which is used to prefix every line of Mosel output with the string `Mosel:` before printing it.

To redirect Mosel streams to a .NET object (.NET `Stream` (including `MemoryStream` for in-memory buffers), `TextReader`, or `TextWriter`) we need to use the `dotnet` I/O driver. The same mechanism that is used here for redirecting the output stream of Mosel (indicated by `XPRMStreamType.F_OUTPUT_LINEBUF` which also enables line buffering) can equally be used to

redirect, for instance, the error stream (denoted by the constant `XPRMStreamType.F_ERROR`).

Note that text read from a `TextReader` will be encoded into bytes via the UTF-8 character encoding before being passed to Mosel; conversely, the text to be written to a `TextWriter` will have been produced by decoding the Mosel output which is assumed to be in UTF-8. If this is not the desired result, consider using a `Stream` instead. If you wish data exchange to be performed in a different encoding (such as the platform's default encoding), this can be done Mosel by use of the `enc` I/O driver (see C.7); only `Streams`, not `TextReaders` or `TextWriters`, are suitable for this.

```
using System;
using System.IO;
using System.Text;
using Mosel;


namespace ugcb.cs {

  public class ugcb {

    /// <summary>
    /// Main entry point for the application
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
      // Initialize Mosel
      XPRM mosel = XPRM.Init();

      // Associate .NET object with a name in Mosel
      mosel.Bind("mycb", new MyOut());

      // Redirect error stream to stdout
      mosel.SetDefaultStream(XPRMStreamType.F_ERROR, Console.Out);

      // Compile and load the Mosel model
      XPRMModel model = mosel.CompileAndLoad("burglar2.mos");

      // Redirect the model's output to a custom TextWriter
      MyOut modelOut = new MyOut();
      model.SetDefaultStream(XPRMStreamType.F_OUTPUT_LINEBUF, modelOut);

      // Alternative:
      // Redirect the model's output to our printing function 'cbmsg'
      model.SetDefaultStream(XPRMStreamType.F_OUTPUT_LINEBUF, "dotnet:mycb");

      // Run the model
      model.Run();

    }
  }

  public class MyOut: TextWriter
  {
   private bool atStartOfLine = true;
   public override void Write(char b)
   {
     if (atStartOfLine) {
      Console.Write("Mosel: ");
      atStartOfLine=false;
     }
     if (b=='\n') {
      Console.WriteLine();
      atStartOfLine=true;
     }
     else if (b=='\r') {
      // ignore
     }
     else {
      Console.Write(b);
```

```
    }
   }
   public override Encoding Encoding {
    get {
     return Encoding.UTF8;
    }
   }
  }
}
```

14.3 Python

The MoselPy package provides a Python interface for Mosel. Install it with `pip install moselpy` and import it as `import moselpy as mp`. Comprehensive coverage of Pandas integration, visualization, iterative solving, and advanced topics is provided in the separate *MoselPy User Guide and Reference Manual*; this section introduces the main features of the interface with the help of small examples.

14.3.1 Compiling and executing a model in Python

With Python, Mosel is initialized automatically when the `moselpy` package is imported. To execute a Mosel model we call the three functions performing the standard compile/load/run sequence as shown in the following example (file `ugcomp.py`).

```
import moselpy as mp

print("Compiling 'burglar2'")
mp.compile_model("burglar2.mos", "burglar2.bim")

print("Loading 'burglar2'")
model = mp.load_model("burglar2.bim")

print("Executing 'burglar2'")
model.run()

print(f"'burglar2' returned: {model.exit_code}")
```

14.3.2 Termination

When running a model in a loop, `reset` is called automatically at the start of each `run` call and does not need to be called explicitly. To release resources after the final run, `reset` can be called manually:

```
model.reset()                          # Free resources after last run
```

This will release all intermediate objects created during the execution without deleting the model itself.

It is also possible to explicitly remove the temporary directory/files created by the execution of Mosel:

```
mp.remove_temp_directory()              # Delete temporary files
```

The `Model` class supports Python's context manager protocol. Using a `with` statement ensures resources are released when the block exits, even if an exception occurs:

```
with mp.load_model("burglar2.bim") as model:
    model.run()
# Model resources automatically released
```

14.3.3 Parameters

When executing a Mosel model in Python, it is possible to pass new values for its parameters into the model. If, for instance, we want to run model 'Prime' from Section 8.3 to obtain all prime numbers up to

500 (instead of the default value 100 set for the parameter `LIMIT` in the model), we may write the following program (file `ugparam.py`):

```
import moselpy as mp

LIM = 500

mp.compile_model("prime.mos", "prime.bim")
model = mp.load_model("prime.bim")

model.run(exec_params={"LIMIT": LIM})

print(f"'prime' returned: {model.exit_code}")
```

Using the MoselPy interface, it is not only possible to compile and run models, but also to access information on the different modeling objects as is shown in the following sections.

14.3.4 Accessing sets

A complete version of a program (file `ugparam.py`) for running the model 'Prime' may look as follows (we work with the model `prime` from Section 8.3 in which all entities are declared as `public`):

```
import moselpy as mp

LIM = 500

mp.compile_model("prime.mos", "prime.bim")
model = mp.load_model("prime.bim")

model.run(exec_params={"LIMIT": LIM})

print(f"'prime' returned: {model.exit_code}")

s_prime = model.find_identifier("SPrime")   # Returns a Python set

if s_prime:
    print(f"Prime numbers from 2 to {LIM}")
    print(" " + ", ".join(str(p) for p in sorted(s_prime)))
```

To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), we retrieve the Mosel object of this name using method `find_identifier`. MoselPy returns Mosel sets as native Python `set` objects, so standard Python operations like `sorted()` and membership tests work directly.

14.3.5 Retrieving solution values

The following program `ugsol.py` executes the model 'Burglar3p' (a variant of 'Burglar3' that stores the decision variable solution values into a `real` array `soltake` via `getsol`) and prints out its solution.

```
import moselpy as mp
import sys

mp.compile_model("burglar3p.mos", "burglar3p.bim")
model = mp.load_model("burglar3p.bim")
model.run()

if not model.prob_status.is_solution_optimal:
    sys.exit(1)                          # Stop if no solution found

print(f"Objective value: {model.objective_value}")

soltake = model.find_identifier("soltake")  # dict: {index: solution_value, ...}
```

```
value = model.find_identifier("VALUE")      # dict: {index: data_value, ...}

for item in sorted(soltake):
    print(f"take({item}): {soltake[item]}\t (item value: {value[item]})")
```

MoselPy returns Mosel arrays as Python dicts, where the keys are the array indices (strings, integers, or tuples for multi-dimensional arrays) and the values are the data entries. To retrieve decision variable solutions, the model stores the `getsol` results into a `public` real array (here `soltake`), which `find_identifier` then returns as a standard dict. There is no need for manual array enumeration.

14.3.6 Sparse arrays

We now again work with the problem 'Transport' that has been introduced in Chapter 3. The flow variables $\mathrm{flow}_{\mathrm{pr}}$ are defined as a *sparse* array. The following example `ugarray.py` prints out all existing entries of the array with their solution values (the model stores these in a `public` real array `solflow` via `getsol`, as explained in Section 14.3.5).

```
import moselpy as mp

mp.compile_model("transportp.mos", "transportp.bim")
model = mp.load_model("transportp.bim")
model.run()

solflow = model.find_identifier("solflow")  # dict with tuple keys

for (plant, region), val in solflow.items():
    print(f"flow({plant},{region}): {val}")
```

MoselPy returns sparse multi-dimensional arrays as dicts with tuple keys. Only defined entries are present in the dict, so iteration naturally covers only the true entries. No explicit enumeration loop is required.

14.3.7 Exchanging data between an application and a model

In the previous examples we have seen how to retrieve information about the model objects from a Mosel model after its execution. In all cases the input data is defined in the model itself or read in from an external data file (`.dat`). However, when embedding a model into an application frequently the input data for the model will be stored (or generated by) the application itself. In this section we therefore show how to pass data in memory from an application to a Mosel model and back, using the `moselpy:` I/O driver.

The MoselPy `moselpy:` driver handles all data exchange through the `input_data` dictionary passed to `run()`. Data returned by the model via `initializations to "moselpy:"` is available in the dict returned by `run()`. This mechanism handles dense, sparse, and scalar data uniformly.

14.3.7.1 Dense arrays

As a first example we look at the case of *dense arrays*. We work with the model `burglar8p.mos`, a version of the Burglar model where all arrays are indexed by the range set `ITEMS = 1..8`. When run standalone, the model reads from `burglard.dat`; when called from Python, the data source is overridden to `moselpy:`. The Mosel model uses the `moselpy:` I/O driver:

```
parameters
 DATASOURCE = 'burglard.dat'  ! overridden to 'moselpy:' from Python
end-parameters

! ...
```

```
 initializations from DATASOURCE
  VALUE  WEIGHT
 end-initializations

 ! ... solve ...

 initializations to DATASOURCE
  soltake
 end-initializations
```

The following program (file `ugiodense.py`) passes the input arrays as dicts with integer keys and receives the solution array in the return value of `run()`:

```
import moselpy as mp

vdata = [15, 100, 90, 60, 40, 15, 10, 1]     # VALUE
wdata = [ 2,  20, 20, 30, 40, 30, 60, 10]    # WEIGHT

mp.compile_model("burglar8p.mos", "burglar8p.bim")
model = mp.load_model("burglar8p.bim")
output = model.run(
    exec_params={"DATASOURCE": "moselpy:"},
    input_data={
        "VALUE":  {i+1: v for i, v in enumerate(vdata)},
        "WEIGHT": {i+1: w for i, w in enumerate(wdata)},
    })

print(f"Objective value: {model.objective_value}")
for i in range(8):
    print(f" take({i+1}): {output['soltake'][i+1]}")
```

The dict keys are matched to Mosel identifiers by name.

14.3.7.2 Sparse arrays

In the previous example data arrays are indexed by a contiguous range (`ITEMS = 1..8`). We now look at the case of data in *sparse format*, where the model `burglar9p.mos` uses a set of strings instead of a simple range set to index the arrays. When run standalone, the model reads from `burglar.dat`; when called from Python, the data source is overridden to `moselpy:`. In the Python program (file `ugiosparse.py`) the dict keys are strings matching the item names:

```
import moselpy as mp

mp.compile_model("burglar9p.mos", "burglar9p.bim")
model = mp.load_model("burglar9p.bim")
output = model.run(
    exec_params={"DATASOURCE": "moselpy:"},
    input_data={
        "VALUE": {"camera": 15, "necklace": 100, "vase": 90,
                  "picture": 60, "tv": 40, "video": 15,
                  "chest": 10, "brick": 1},
        "WEIGHT": {"camera": 2, "necklace": 20, "vase": 20,
                   "picture": 30, "tv": 40, "video": 30,
                   "chest": 60, "brick": 10},
    })

print(f"Objective value: {model.objective_value}")
for item, val in output["soltake"].items():
    print(f" take({item}): {val}")
```

Mosel populates the set `ITEMS` automatically from the keys of the incoming data. Explicit set initialization from Python is not required.

14.3.7.3 Scalars

Scalars are exchanged in the same way as arrays: simply pass a Python `int`, `float`, or `str` value directly in the `input_data` dict:

```
output = model.run(
    exec_params={"DATASOURCE": "moselpy:"},
    input_data={"WTMAX": 100})

print(f"Objective value: {output['ObjVal']}")
print(f"Total items: {output['NumItem']}")
```

Alternatively, scalar values may be passed as model parameters via `exec_params` (as shown in Section 14.3.3).

14.3.8 Redirecting the Mosel output

When executing a Mosel model from a Python application it may be desirable to suppress or capture the output produced by Mosel. The `set_default_stream()` method redirects Mosel streams. The following program (file `ugcb.py`) shows how to capture all Mosel output into a Python buffer and prefix every line before printing it:

```
import moselpy as mp
from io import StringIO

mp.compile_model("burglar2.mos", "burglar2.bim")
model = mp.load_model("burglar2.bim")

# Capture output to a Python buffer
buf = StringIO()
model.symbols["out"] = buf
model.set_default_stream(mp.StreamType.OUTPUT, "moselpy:out")
model.run()
del model.symbols["out"]

captured = buf.getvalue()
for line in captured.splitlines():
    print(f"Mosel: {line}")
```

To simply suppress all output, redirect to `null:` instead:

```
model.set_default_stream(mp.StreamType.OUTPUT, "null:")
```

The same mechanism that is used here for redirecting the output stream of Mosel (indicated by `StreamType.OUTPUT`) can equally be used to redirect, for instance, the error stream (denoted by the constant `StreamType.ERROR`).

14.4 VBA

VBA typically serves for embedding a Mosel model into an Excel spreadsheet. In this section we shall only show the parts relevant to the Mosel functions, assuming that the execution of a model is trigged by the action of clicking on some object such as the buttons shown in Figure 14.1.

14.4.1 Compiling and executing a model in VBA

As with the other programming languages, to execute a Mosel model in VBA we need to perform the standard compile/load/run sequence as shown in the following example (contained in the file `ugvb.bas`). We use a slightly modified version `burglar5.mos` of the burglar problem where we have redirected the output printing to the file `burglar_out.txt`.

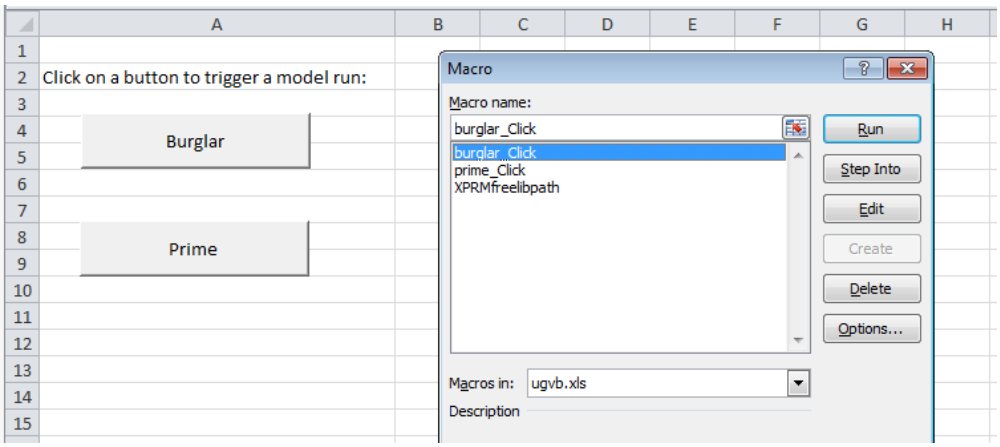


Figure 14.1: Excel spreadsheet embedding VBA macros

```
Private Sub burglar_Click()
  Dim model
  Dim ret As Long
  Dim result As Long
  Dim outfile As String, moselfile As String

'Initialize Mosel
  ret = XPRMinit
  If ret Then
    MsgBox "Initialization error (" & ret & ")"
    Exit Sub
  End If

'Compile burglar5.mos
  XPRMsetdefworkdir GetFullPath()
  moselfile = GetFullPath() & "\" & "burglar5"
  outfile = GetFullPath() & "\" & "burglar_out.txt"
  ret = XPRMcompmod(vbNullString, moselfile & ".mos", vbNullString, _
        "Burglar problem")
  If ret <> 0 Then
    MsgBox "Compile error (" & ret & ")"
    Exit Sub
  End If

'Load burglar5.bim
  model = XPRMloadmod(moselfile & ".bim", vbNullString)
  If model = 0 Then
    MsgBox "Error loading model"
    Exit Sub
  End If

'Run the model
  ret = XPRMrunmod(model, result, "OUTFILE=""" & Replace(outfile, "\", _
        "\\") & """")
  If ret <> 0 Then
    MsgBox "Execution error (" & ret & ")"
    GoTo done
  Else
    ShowFile outfile
  End If
  MsgBox vbNewLine & "model Burglar returned: " & result

done:
  XPRMfree
End Sub

'Auxiliary routines
```

```
Private Sub ShowFile(fn As String)
  Dim vs As String
  vs = CreateObject("Scripting.FileSystemObject").OpenTextFile(fn).ReadAll
  MsgBox vs
End Sub

Private Function GetFullPath() As String
  Dim path As String
  path = ThisWorkbook.path
  If Right(path, 1) = "\" Then path = Left(path, Len(path) - 1)
  GetFullPath = path
End Function
```

This implementation redirects the output into a log file `vbout.txt` the contents of which is displayed after a successful model run.

14.4.2 Parameters

When executing a Mosel model in VBA, it is possible to pass new values for its parameters into the model. The following program (also contained in the file `ugvb.frm`) extract shows how we may run model 'Prime' from Section 8.3 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model). We use a slightly modified version `prime4.mos` of the model where we have redirected the output printing to a file denoted by the parameter `OUTFILE`.

```
Private Sub prime_Click()
  Dim model
  Dim ret As Long
  Dim result As Long
  Dim outfile As String, moselfile As String

'Initialize Mosel
  ret = XPRMinit
  If ret Then
    MsgBox "Initialization error (" & ret & ")"
    Exit Sub
  End If

'Compile prime4.mos
  XPRMsetdefworkdir GetFullPath()
  moselfile = GetFullPath() & "\" & "prime4"
  outfile = GetFullPath() & "\" & "vbout.txt"
  ret = XPRMcompmod(vbNullString, moselfile & ".mos", vbNullString, _
        "Prime numbers")
  If ret <> 0 Then
    MsgBox "Compile error (" & ret & ")"
    Exit Sub
  End If

'Load prime4.bim
  model = XPRMloadmod("prime4.bim", vbNullString)
  If model = 0 Then
    MsgBox "Error loading model"
    Exit Sub
  End If

'Run model with new parameter settings
  ret = XPRMrunmod(model, result, "LIMIT=500,OUTFILE=""" & Replace(outfile, _
        "\", "\\") & """")

  If ret <> 0 Then
    MsgBox "Execution error (" & ret & ")"
    GoTo done
  Else
    ShowFile outfile
  End If
```

```
  MsgBox vbNewLine & "model Prime returned: " & result

done:
  XPRMfree
End Sub
```

14.4.3 Redirecting the Mosel output

In the previous example we have hardcorded the redirection of the output directly in the model. With Mosel's VBA interface the user may also redirect all output produced by Mosel to files directly from the host application, that is, redirect the output stream.

To redirect all output of a model to the file `myout.txt` add the following function call before the execution of the Mosel model:

```
' Redirect all output to the file "myout.txt"
 XPRMsetdefstream 0, XPRM_F_OUTPUT, "myout.txt"
```

Similarly, any possible error messages produced by Mosel can be recovered by replacing in the line above `XPRM_F_OUTPUT` by `XPRM_F_ERROR`. This will redirect the error stream to the file `myout.txt`.

The following VBA program extract (file `ugcb.bas`) shows how to use a callback in VBA to receive all output from a Mosel model (standard output and errors). The output will be displayed in the spreadsheet from where the model run was started.

```
Private ROWNUM As Long
Public Sub example()
  Dim ret As Long
  Dim result As Long
  Dim module

  ClearColumn

' Initialize Mosel. Must be called first
  ret = XPRMinit
  If ret <> 0 Then
    PrintLn ("Failed to initialize Mosel")
    Exit Sub
  End If

' Redirect the output and error streams to the callback
  ret = XPRMsetdefstream(0, XPRM_F_OUTPUT, XPRM_IO_CB(AddressOf OutputCB))
  ret = XPRMsetdefstream(0, XPRM_F_ERROR, XPRM_IO_CB(AddressOf OutputCB))

  PrintLn "Starting model..."

' Run the model
  ret = XPRMexecmod("", GetFullPath() & "\" & "burglar10.mos", _
                    "FULLPATH='" & GetFullPath() & "'", result, module)
  If ret <> 0 Then
    PrintLn ("Failed to execute model")
    GoTo done
  Else
    PrintLn "Finished model"
  End If

  done:
    XPRMfree
End Sub

#If VBA7 Then
Private Sub OutputCB(ByVal model As LongPtr, ByVal ref As LongPtr, _
                     ByVal msg As String, ByVal size As Long)
  ' Output to the spreadsheet
```

```
  Call PrintLn(msg)
End Sub
#Else
Private Sub OutputCB(ByVal model As Long, ByVal ref As Long, _
                     ByVal msg As String, ByVal size As Long)
  ' Output to the spreadsheet
  Call PrintLn(msg)
End Sub
#End If

Public Sub PrintLn(ByVal msg As String)
  ' Strip any trailing newlines first
  If Right(msg, Len(vbLf)) = vbLf Then msg = Left(msg, Len(msg) - Len(vbLf))
  If Right(msg, Len(vbCr)) = vbCr Then msg = Left(msg, Len(msg) - Len(vbCr))
  Worksheets("Run Model").Cells(ROWNUM, 2) = Trim(msg)
  ROWNUM = ROWNUM + 1
End Sub

Sub ClearColumn()
  Worksheets("Run Model").Columns(2).ClearContents
  ROWNUM = 1
End Sub

Function GetFullPath() As String
  Dim path As String
  path = ThisWorkbook.path
  If Right(path, 1) = "\" Then path = Left(path, Len(path) - 1)
  GetFullPath = path
End Function
```

IV. Extensions and tools

Overview

Beyond what one might call the 'standard use' of Mosel, the Mosel environment has an increasing number of advanced features, some of which you might find helpful for the development or deployment of larger applications.

The first chapter of this part (Chapter 15) introduces the Mosel Debugger and Profiler, two tools that are particularly helpful for the development and analysis of large-scale Mosel models. We give some hints how you might improve the efficiency of your models.

The next chapter (Chapter 16) introduces the notion of *packages* and shows several examples of their use. It also discusses the differences between packages and modules and their respective uses.

Chapter 17 gives an overview of other advanced functionality, including generalized file handling, concurrency in modeling, graphing, and other solver types. In depth introductions to these topics are given in separate manuals or whitepapers to avoid overloading this user guide.

The last chapter introduces the notion of 'annotations' and tools of the Mosel distribution that exploit such metadata, for example in the automated generation of Mosel model documentation.

CHAPTER 15

Debugger and Profiler

15.1 The Mosel Debugger

In Chapter 6 we have seen how the Mosel Parser helps detect syntax errors during compilation. Other types of errors that are in general more difficult to analyze are mistakes in the data or logical errors in the formulation of Mosel models. The Mosel Debugger may help tracing these.

15.1.1 Using the Mosel Debugger

In this section we shall be working with the model `prime2.mos`. This is the same model for calculating prime numbers as the example we have seen in Section 8.3, but with a `LIMIT` value set to 20,000.

Mosel models that are to be run in the debugger need to be compiled with the option `G`. The Mosel debugger is started with the command `debug` (it will automatically compile with the required settings):

```
mosel debug prime.mos
```

and terminated by typing `quit`. Just as for the `run` command the user may specify new settings for the model parameters immediately following the `debug` command:

```
mosel debug prime2.mos 'LIMIT=50'
```

Once the debugger is started, type in the following sequence of commands (Mosel's output is highlighted in bold face):

```
dbg>break 31
Breakpoint 1-1 set at prime2.mos:31
dbg>bcond 1-1 getsize(SNumbers) <10
dbg>cont
Prime numbers between 2 and 50:
Breakpoint 1-1.
31 while (not(n in SNumbers)) n+=1
dbg>print n
13
dbg>display SNumbers
1(0): SNumbers = [17 19 23 29 31 37 41 43 47]
dbg>display SPrime
2(0): SPrime = [2 3 5 7 11 13]
dbg>cont
Breakpoint 1-1.
31 while (not(n in SNumbers)) n+=1
1(0): SNumbers = [19 23 29 31 37 41 43 47]
2(0): SPrime = [2 3 5 7 11 13 17]
dbg>cont
Breakpoint 1-1.
31 while (not(n in SNumbers)) n+=1
1(0): SNumbers = [23 29 31 37 41 43 47]
2(0): SPrime = [2 3 5 7 11 13 17 19]
dbg>cont
Breakpoint 1-1.
31 while (not(n in SNumbers)) n+=1
1(0): SNumbers = [29 31 37 41 43 47]
2(0): SPrime = [2 3 5 7 11 13 17 19 23]
dbg>quit
```

This small example uses many of the standard debugging commands (for a complete list, including commands for navigating in the Mosel stack that are not shown here, please see the Section 'Running Mosel – Command line interpreter: debugger' of the introduction chapter of the Mosel Language Reference Manual):

`break`	Set a breakpoint in the given line. A breakpoint is deleted with `delete` followed by the breakpoint number. The command `breakpoints` lists all currently defined breakpoints.
`bcond`	Set a condition on a breakpoint (using the number of the breakpoint returned by the `break` command). Conditions are logical expressions formed according to the standard rules in Mosel (use of brackets, connectors `and` and `or`). They may contain any of the functions listed below.
`cont`	Continue the execution up to the next breakpoint (or to the end of the program). A line-wise evaluation is possible by using `next` or `step` (the former jumps over loops and subroutines, the latter steps into them).
`display`	Show the current value of a model object or an expression at every step of the debugger. A display is removed by calling `undisplay` followed by the number of the display.
`print`	Show (once) the current value of a model object.

The following simple Mosel functions may be used with debugger commands (in conditions or with `print` / `display`):

- Arithmetic functions: `abs`, `ceil`, `floor`, `round`
- Accessing solution values: `getsol`, `getdual`, `getrcost`, `getactivity`, `getslack`
- Other: `getparam`, `getsize`

15.1.1.1 Debugging concurrent models

The Mosel debugger can be used with concurrent (sub)models. A few temporary edits to the model files may be necessary in this case (to be removed for production versions!):

- We need to use the compilation flag 'G' with all models that are to be debugged: this option is applied automatically by the Mosel debugger for the main model, but we need to use it explicitly for the submodels. For instance, if the submodels are compiled from another model, we need to modify the compilation statement to include this flag:

```
if compile("G","prime2d.mos")<>0: exit(1)
```

- Breakpoints on submodels can only be set once the corresponding submodels have been started. If their execution is too fast to allow for user input in the debugger, we recommend to insert a 'sleep' at the start of the submodel (this subroutine is provided by the module *mmsystem* that needs to be loaded by the model) during the debugging phase. For example adding a 'sleep' of 5 seconds:

```
sleep(5000)
```

We can now start the debugger for the main model with a command like the following:

```
mosel debug runprime2d.mos 'LIMIT=50'
```

Once the debugger is started, type in the following sequence of commands (Mosel's output is highlighted in bold face):

```
dbg>break 29
Breakpoint 1-1 set at runprime2d.mos:29
dbg>cont
29 run(modPrime, "LIMIT="+LIMIT)
dbg>next
30 wait
[model #2 starting]
dbg>model 2
* "prime2d.mos":18
18 LIMIT=20000
dbg>break 34
Breakpoint 2-1 set at prime2d.mos:34
dbg>bcond 2-1 getsize(SNumbers) <10
dbg>cont
Prime numbers between 2 and 50:
Breakpoint 2-1.
34 while (not(n in SNumbers)) n+=1
dbg>print n
13
dbg>display SNumbers
1(#1): SNumbers = [17 19 23 29 31 37 41 43 47]
dbg>display SPrime
2(#1): SPrime = [2 3 5 7 11 13]
dbg>cont
Breakpoint 2-1.
34 while (not(n in SNumbers)) n+=1
1(#1): SNumbers = [19 23 29 31 37 41 43 47]
2(#1): SPrime = [2 3 5 7 11 13 17]
dbg>cont
Breakpoint 2-1.
...
dbg>quit
```

This sequence sets a breakpoint on the submodel 'run' command in the main model. After the submodel is started it switches to using the submodel ('model 2') as the active model in the debugger; we can then

enter the debug commands for the submodel. Notice that the command `cont` applies globally to all running models, whereas `next` or `step` refer to the selected active model.

15.1.2 Debugger in Xpress Workbench

With Xpress Workbench the debugger is started by clicking on the button . Workbench will automatically recompile the model with the required debugging flag. Breakpoints are set by clicking onto the gray area (left to the line number if it is displayed) preceding each row in the editor window, breakpoint conditions can be added via the right mouse button menu on the breakpoint icon. Clicking on the breakpoint icon deletes the breakpoint.



Delete breakpoint/deactivated breakpoint.

Delete a conditional breakpoint/deactivated conditional breakpoint.

Navigating in the debugger is possible by clicking on the corresponding buttons:



Activate/deactivate all breakpoints.

Start/stop the debugger.

Resume/suspend model execution.

Step over an expression.

Step into an expression.

Step out of an expression.

During a debugging session, the current position is indicated via a green arrow left to the line numbers  (changing to yellow on breakpoints ). Expand the *Variables* display in the *Debugger* pane on the right side of the workspace to observe the values of model entities.

15.2 Efficient modeling through the Mosel Profiler

The efficiency of a model may be measured through its *execution speed* and also its *memory consumption*. The execution times can be analyzed in detail with the help of the Mosel Profiler. Several commands of the Mosel debugger that are also discussed in this section provide the user with further information, such as memory consumption.

15.2.1 Using the Mosel Profiler

Once a model you are developing is running correctly, you will certainly start testing it with larger data sets. Doing so, you may find that model execution times become increasingly larger. This may be due to the solution algorithms, but a more or less significant part of the time will be spent simply in defining the model itself. The Mosel Profiler lets you analyze the model behavior line-by-line. Solution algorithms, such as LP or MIP optimization with Xpress Optimizer, may be improved by tuning solver parameters (please refer to the corresponding software manuals). Here we shall be solely concerned with improvements that may be made directly to the Mosel model. Even for large scale models, model execution times can often be reduced to just a few seconds by carefully (re)formulating the model.

Just as for the debugger, Mosel models that are to be run in the profiler need to be compiled with the option `G`. The command `profile` of the Mosel command line performs all required steps:

```
mosel profile prime2.mos
```

or, if we wish to generate the BIM file explicitly:

```
mosel comp -G prime2.mos
mosel run -prof prime2.bim
```

The profiler generates a file *filename*`.prof` with the profiling statistics. For the test model `prime2.mos` this file has the following contents (leaving out the header lines and comments):

```
                          model Prime

                           parameters
      1     0.00     0.00   LIMIT=20000
                           end-parameters

                           declarations
      1     0.00     0.00   SNumbers: set of integer
      1     0.00     0.00   SPrime: set of integer
                           end-declarations

      1     0.01     0.00  SNumbers:={2..LIMIT}

      1     0.00     0.01  writeln("Prime numbers between 2 and ", LIMIT, ":")

      1     0.00     0.01  n:=2
      1     0.00     0.01  repeat
   2262     0.04     3.44    while (not(n in SNumbers)) n+=1
   2262     0.00     3.44    SPrime += {n}
   2262     0.00     3.44    i:=n
   2262     0.04     3.44    while (i<=LIMIT) do
  50126     3.31     3.44      SNumbers-= {i}
  50126     0.04     3.44      i+=n
                             end-do
   2262     0.00     3.44  until SNumbers={}

      1     0.00     3.44  writeln(SPrime)
      1     0.00     3.45  writeln(" (", getsize(SPrime), " prime numbers.)")

      1     0.00     3.45 end-model
```

The first column lists the number of times a statement is executed, the second column the total time spent in a statement, and the third column the time of the last execution; then follows the corresponding model statement. In our example, we see that most of the model execution time is spent in a single line, namely the deletion of elements from the set `SNumbers`. This line is executed more than 50,000 times, but so is the following statement (`i+=n`) and it only takes a fraction of a second. Indeed, operations on large (>1000 entries) sets may be relatively expensive in terms of running time. If our prime number algorithm were to be used in a large, time-critical application we should give preference to a more suitable data structure that is addressed more easily, that is, an array. For instance, by modifying the model as follows the total execution time for this model version becomes 0.19 seconds:

```
model "Prime (array)"

 parameters
  LIMIT=20000                  ! Search for prime numbers in 2..LIMIT
 end-parameters

 declarations
  INumbers = 2..LIMIT          ! Set of numbers to be checked
  SNumbers: array(INumbers) of boolean
  SPrime: set of integer       ! Set of prime numbers
 end-declarations

 writeln("Prime numbers between 2 and ", LIMIT, ":")

 n:=2
 repeat
   SPrime += {n}               ! n is a prime number
   i:=n
   while (i<=LIMIT) do         ! Remove n and all its multiples
```

```
    SNumbers(i):= true
    i+=n
  end-do
  while (n <= LIMIT and SNumbers(n)) n+=1
until (n>LIMIT)

writeln(SPrime)
writeln(" (", getsize(SPrime), " prime numbers.)")

end-model
```

To start the Mosel profiler from Xpress Workbench, open the *Run Dialog* window from the menu *Run* or by clicking on the tools button ▾ and select the run mode *Profile the Model*.

15.2.1.1 Profiling concurrent models

The Mosel profiler can be used to profile models that are running concurrently. The profiler run is started by launching the profiler for the main (root) model. For every model file, an output file *filename*`.prof` is generated. If several instances of the same model file are being run, Mosel creates unique filenames of the form *filename*`.`*modelid*`.prof` where *modelid* is formed from the model counter per model tree level.

The user is reminded that all (sub)models used in profiler runs need to be compiled with the 'G' flag.

15.2.2 Other commands for model analysis

The Mosel debugger provides a few other commands that may be helpful with quickly obtaining information about models that have been executed in Mosel.

Consider, for example, the following model `flow.mos`.

```
model "Dynamic arrays"
 declarations
  Suppliers = 1..150
  Customers = 1..10000
  COST: dynamic array(Suppliers,Customers) of real
  flow: dynamic array(Suppliers,Customers) of mpvar
 end-declarations

 initializations from "flow.dat"
  COST
 end-initializations

 forall(s in Suppliers, c in Customers | COST(s,c)>0 ) create(flow(s,c))

end-model
```

Now execute the following sequence of Mosel commands from the command line (as before, Mosel output is printed in bold face). The commands we wish to use are part of the Mosel debugger.—Since we do not wish to launch a debugging session, we use the option `-g` to compile in debug mode, but without tracing information. This results in a standard model run without entering an interactive debugging session.

```
mosel debug -g flow.mos
dbg>lsmods
* name: Dynamic arrays (0.0.0) number: 1 size: 47884
sys. com.: `flow.mos',debug,mc5.0.0
user com.:
dbg>info COST
`COST' is an array (dynamic, dim: 2, size: 750) of reals
dbg>quit
```

The command `lsmods` displays information about all models loaded in Mosel, and in particular their size (= memory usage in bytes). With the command `info` followed by a symbol name we obtain detailed information about the definition of this symbol (without giving a symbol this command will display release and license information for Mosel). Alternatively, it is also possible to print the complete list of symbols (with type information and sizes) defined by the current model by using the command `lssymb`.

If we now remove the keyword `dynamic` from the declaration of the two arrays, `COST` and `flow`, and re-run the same command sequence as before, we obtain the following output:

```
dbg>lsmods
* name: Dynamic arrays number: 1 size: 36011152
Sys. com.: `flow.mos',debug,mc5.0.0
User com.:
dbg>info COST
`COST' is an array (dim: 2, size: 1500000) of reals
```

We can run a similar experiment with the model version `flowh.mos` that defines the two sparse arrays as `hashmap`. As shown in the output below, the memory usage is somewhat higher albeit in the same order of magnitude as the model version with dynamic arrays:

```
mosel debug -g flowh.mos
dbg>lsmods
* name: Hashmap arrays (0.0.0) number: 1 size: 81488
sys. com.: `flowh.mos',debug,mc5.0.0
user com.:
dbg>info COST
`COST' is an array (hashmap, dim: 2, size: 750) of reals
dbg>quit
```

It is easily seen that in this model the use of the keyword `dynamic` or `hashmap` makes a huge difference in terms of memory usage. A model defining several arrays of comparable sizes is likely to run out of memory (or at the least, it may not leave enough memory for an optimization algorithm to be executed).

Note: If `COST` is defined as a sparse (dynamic or hashmap) array, the condition on the `forall` loop should really be `exists(COST(s,c))` for speedier execution of this loop.

15.2.3 Some recommendations for efficient modeling

The following list summarizes some crucial points to be taken into account, especially when writing large-scale models. For more details and examples please see Appendix B.

- Use sparse arrays to
  - size data tables automatically when the data is read in,
  - initialize the index values automatically when the data is read in,
  - conserve memory when storing sparse data,
  - eliminate index combinations without using conditions each time.
- Don't use sparse arrays
  - when you can use ordinary (dense) arrays instead,
  - when storing dense data (if at least 50% of its entries are defined an array should clearly be considered as dense), and you can size the data table and initialize the indices in some other way, such as by reading in the size first.
- General procedure for declaring and initializing data:

1. declare all index sets and the minimal collection of data arrays required to initialize the sets,
2. initialize the data arrays (which also initializes all index sets),
3. finalize the index sets,
4. declare and initialize all other arrays.

- Efficient use of sparse arrays:
  - use the keyword `exists` for enumeration (in sums or loops),
  - access the elements in *ascending order* of the indices (particularly with `dynamic` arrays),
  - use `hashmap` when array elements are predominantly accessed in random order,
  - use *ranges*, rather than sets, for the index sets.
- Efficient use of `exists`:
  - use *named index sets* in the declarations,
  - use *the same index sets* in the loops,
  - use the index sets in the *same order*,
  - use the `dynamic`/`hashmap` qualifier if some index sets are constant or finalized,
  - make sure `exists` is the *first condition*,
  - always use `exists`, even if no condition or an alternative condition is logically correct,
  - conditions with `or` cannot be handled as efficiently as conditions with `and`.
- Loops (`sum`, `forall`, *etc.*):
  - where possible, use conditional loops—loop index set followed by a vertical bar and the condition(s)—instead of a logical test with `if` within the loop,
  - make sure `exists` is the *first condition*,
  - always use `exists`, even if no condition or an alternative condition is logically correct,
  - enumerate the index sets in the *same order* as they occur in the arrays within the loop,
  - broken up, conditional loops are handled less efficiently.
- Do not use any debugging flag for compiling the final deployment version of your models.

CHAPTER 16

Packages

A *package* is a library written in the Mosel language (this feature was introduced by Mosel 2.0). Its structure is similar to models, replacing the keyword `model` by `package`. Packages are included into models with the `uses` statement for dynamic loading (the package BIM needs to be present for model execution), in the same way as this is the case for modules (DSO). Alternatively, packages can be loaded statically via `imports` in which case they get included in the model BIM file (this option is not available for modules that are always dynamic). Unlike Mosel code that is included into a model with the `include` statement, packages are compiled separately, that is, their contents are not visible to the user.

Typical uses of packages include

- development of your personal 'tool box'
- model parts (*e.g.*, reformulations) or algorithms written in Mosel that you wish to distribute without disclosing their contents
- add-ons to modules that are more easily written in the Mosel language

Packages may define new constants, subroutines, types, and parameters for the Mosel language as shown in the following examples (the first two examples correspond to the first two module examples of the Mosel Native Interface User Guide).

Please note the following:

1. **Package name:** compiling a package will result in a file *`packagename`*`.bim`. This package is invoked in a Mosel model by the statement
   `uses "`*`packagename`*`"`
   The name of the Mosel package source file (`.mos` file) may be different from the name given to the BIM file.
2. **Internal package name:** the name given in the Mosel file after the keyword `package` is the internal name of the package. It must be a valid Mosel identifier (and not a string). This name may be different from the name given to the BIM file, but it seems convenient to use the same name for both.
3. **Package location:** for locating packages Mosel applies similar rules as for locating modules; it first uses the list of prefixes defined by the compiler option `–bx` or the environment variable `MOSEL_BIM`, it then searches in the directory `dso` of the Xpress installation (see Section *2.3.1 Directive uses* of the Mosel Language Reference for further detail), and in the directories pointed to by the environment variable `MOSEL_DSO`. The contents of the enviroment variables `MOSEL_BIM` and `MOSEL_DSO` can be set freely by the user.
   To try out the package examples in this chapter, you may simply include the current working directory (`'.'`) in the locations pointed to by `MOSEL_DSO`, so that packages in the current working directory will be found, for example:
   Windows: `set MOSEL_DSO=.`

Unix/Linux, C shell: `setenv MOSEL_DSO .`
Unix/Linux, Bourne shell: `export MOSEL_DSO; MOSEL_DSO=.`

Alternatively, you can use the *compilation option* `-bx` to indicate the location of package files (this option does not apply to DSOs):

```
mosel exe -bx ./ mymodel.mos
```

In general, and in particular for the deployment of an application, it is recommended to work with absolute paths in the definition of environment variables.

16.1 Definition of constants

The following package `myconstants` defines one integer, one real, one string, and two boolean constants.

```
package myconstants

 public declarations
  MYCST_BIGM = 10000          ! A large integer value
  MYCST_TOL = 0.00001         ! A tolerance value
  MYCST_LINE =                ! String constant
     "-----------------------------------------------------------------"
  MYCST_FLAG = true           ! Constant with value true
  MYCST_NOFLAG = false        ! Constant with value false
 end-declarations

end-package
```

The structure of a package is similar to the structure of Mosel models, with the difference that we use the keyword `package` instead of `model` to mark its beginning and end.

After compiling our package with the standard Mosel command (assuming the package is saved in file `myconstants.mos`)

```
mosel comp myconstants
```

it can be used in a Mosel model (file `myconst_test.mos`):

```
model "Test myconstants package"
 uses "myconstants"

 writeln(MYCST_LINE)
 writeln("BigM value: ", MYCST_BIGM, ", tolerance value: ", MYCST_TOL)
 writeln("Boolean flags: ", MYCST_FLAG, " ", MYCST_NOFLAG)
 writeln(MYCST_LINE)

end-model
```

Having made sure that Mosel is able to find our package `myconstants.bim` (see the notes about package location at the beginning of this chapter), executing the test model above will produce the following output:

```
-----------------------------------------------------------------
BigM value: 10000, tolerance value: 1e-05
Boolean flags: true false
-----------------------------------------------------------------
```

When comparing with the C implementation of the module example `myconstants` in the Mosel Native Interface User Guide we can easily see that the package version is much shorter.

16.2 Definition of subroutines

We now show a package (file `solarraypkg.mos`) that defines several versions of a subroutine, `solarray`, which copies the solution values of an array of decision variables of type `mpvar` into an array of real of the same size. For each desired number (1–3) and type (integer or string) of array indices we need to define a new version of this subroutine.

```
package solarraypkg

! **** Integer indices (including ranges) ****
 public procedure solarray(x:array(R:set of integer) of mpvar,
                           s:array(set of integer) of real)
  forall(i in R) s(i):=getsol(x(i))
 end-procedure

 public procedure solarray(x:array(R1:set of integer,
                                   R2:set of integer) of mpvar,
                           s:array(set of integer,
                                   set of integer) of real)
  forall(i in R1, j in R2) s(i,j):=getsol(x(i,j))
 end-procedure

 public procedure solarray(x:array(R1:set of integer,
                                   R2:set of integer,
                                   R3:set of integer) of mpvar,
                           s:array(set of integer,
                                   set of integer,
                                   set of integer) of real)
  forall(i in R1, j in R2, k in R3) s(i,j,k):=getsol(x(i,j,k))
 end-procedure

! ****String indices ****
 public procedure solarray(x:array(R:set of string) of mpvar,
                           s:array(set of string) of real)
  forall(i in R) s(i):=getsol(x(i))
 end-procedure

 public procedure solarray(x:array(R1:set of string,
                                   R2:set of string) of mpvar,
                           s:array(set of string,
                                   set of string) of real)
  forall(i in R1, j in R2) s(i,j):=getsol(x(i,j))
 end-procedure

 public procedure solarray(x:array(R1:set of string,
                                   R2:set of string,
                                   R3:set of string) of mpvar,
                           s:array(set of string,
                                   set of string,
                                   set of string) of real)
  forall(i in R1, j in R2, k in R3) s(i,j,k):=getsol(x(i,j,k))
 end-procedure
end-package
```

Using the package in a Mosel model (file `solarrpkg_test.mos`):

```
model "Test solarray package"
 uses "solarraypkg", "mmxprs"

 declarations
  R1=1..2
  R2={6,7,9}
  R3={5,-1}
  x: array(R1,R2,R3) of mpvar
  sol: array(R1,R2,R3) of real
 end-declarations
```

```
! Define and solve a small problem
 sum(i in R1, j in R2, k in R3) (i+j+2*k) * x(i,j,k) <= 20
 forall(i in R1, j in R2, k in R3) x(i,j,k)<=1
 maximize(sum(i in R1, j in R2, k in R3) (i+2*j+k) * x(i,j,k))

! Get the solution array
 solarray(x,sol)

! Print the solution
 forall(i in R1, j in R2, k in R3)
  writeln(" (", i, ",", j, ",", k, ") ", sol(i,j,k), " ", getsol(x(i,j,k)))
 writeln(sol)

end-model
```

Output produced by this model:

```
 (1,6,-1) 1 1
 (1,6,5) 0 0
 (1,7,-1) 1 1
 (1,7,5) 0 0
 (1,9,-1) 1 1
 (1,9,5) 0 0
 (2,6,-1) 0.166667 0.166667
 (2,6,5) 0 0
 (2,7,-1) 0 0
 (2,7,5) 0 0
 (2,9,-1) 0 0
 (2,9,5) 0 0
[1,0,1,0,1,0,0.166667,0,0,0,0,0]
```

This example may be classified as a 'utility function' that eases tasks occurring in a similar way in several of your models. Another example of such a utility function could be a printing function that simply outputs the solution value of a decision variable with some fixed format (if you apply `write`/`writeln` to a decision variable of type `mpvar` you obtain the pointer of the variable, and not its solution value).

If we again make the comparison with the implementation as a module we see that both ways have their positive and negative points: the implementation as a module is clearly more technical, requiring a considerable amount of C code not immediately related to the implementation of the function itself. However, at the C level we simply check that the two arguments have the same index sets, without having to provide a separate implementation for every case, thus making the definition more general.

16.2.1 Implementation alternative using union types and iterator

A generic implementation of the subroutine `solarray` (see file `solarranypkg.mos`) can be achieved with the help of union types in combination with the `iterator` functionality provided by the module *mmreflect* that makes it possible to enumerate arrays of dense or sparse format without any explicit knowledge about their indexing sets. As a first step, this routine needs to check whether the two arguments are indeed arrays of the expected types (`mpvar` and `real` respectively), followed by a comparison of the index sets, similarly to the C implementation as a module. This new implementation accepts any type and number of index sets and is therefore considerably more generic than any set of explicitly defined overloaded subroutine versions.

```
package solarrayanypkg
 uses "mmreflect"

 ! Utility routine checking whether 2 arrays have the same index set types
 function checkindices(x:any, s:any): boolean
   returned:= x.array.nbdim=s.array.nbdim and
     and(i in 1.. x.array.nbdim)
       x.array.index(i).eltype=s.array.index(i).eltype
 end-function
```

```
 ! Generic implementation of solarray routine
 public procedure solarray(x:any, s:any)
   declarations
     it: iterator
   end-declarations
   if x is array of mpvar and s is array of real then
     if checkindices(x,s) then
       initiiter(it,x.array)
       reset(s.array)
       while (nextcell(it)) s.array(it).real:= x.array(it).mpvar.sol
     else
       writeln("SOLARRAY: array indices don't match")
     end-if
   else
     writeln("SOLARRAY: arguments must be arrays of type mpvar / real")
   end-if
 end-procedure

end-package
```

This subroutine `solarray` provided by the package `solarranypkg` is used exactly in the same way as what we have seen earlier for the package `solarrpkg`.

16.3 Definition of types

In Section 8.6.2 we have seen the example `arcs.mos` that defines a record to represent arcs of a network. If we wish to use this data structure in different models we may move its definition into a package 'arcpkg' to avoid having to repeat it in every model.

Such a package may look as follows (file `arcpkg.mos`):

```
package arcpkg

 public declarations
  arc = public record                  ! Arcs:
   Source,Sink: string                 !   Source and sink of arc
   Cost: real                          !   Cost coefficient
  end-record
 end-declarations

end-package
```

which is used thus from the model file:

```
model "Arcs2"
 uses "arcpkg"

 declarations
  NODES: set of string                 ! Set of nodes
  ARC: array(ARCSET:range) of arc      ! Arcs
 end-declarations

 initializations from 'arcs.dat'
  ARC
 end-initializations

! Calculate the set of nodes
 NODES:=union(a in ARCSET) {ARC(a).Source, ARC(a).Sink}
 writeln(NODES)

 writeln("Average arc cost: ", sum(a in ARCSET) ARC(a).Cost / getsize(ARCSET) )

end-model
```

At this place, the use of the keyword `public` may call for some explanation. Here and also in the example 'myconstants' the whole `declarations` block is preceded by the `public` marker, indicating that all objects declared in the block are public (*i.e.*, usable outside of the package definition file). If only some declarations are public and others in the same block are private to the package, the `public` marker needs to preceed the name of every object within the declarations that is to become public instead of marking the entire block as public.

The second occurrence of the `public` marker in the definition of package 'arcpkg' is immediately in front of the keyword `record`, meaning that all fields of the record are public. Again, it is possible to select which fields are accessible from external files (for example, you may wish to reserve some fields for special flags or calculations within your package) by moving the keyword `public` from the record definition in front of every field name that is to be marked as public.

A definition of package 'arcpkg' equivalent to the one printed above therefore is the following.

```
package arcpkg2

 declarations
  public arc = record                  ! Arcs:
   public Source,Sink: string          !   Source and sink of arc
   public Cost: real                   !   Cost coefficient
  end-record
 end-declarations

end-package
```

16.4 Definition of parameters

Mosel parameters are scalars of one of the four basic types (real/integer/string/boolean). Packages can define new parameters by declaring their names and type in the `parameters` section. The package needs to store the current values of the parameters in separate model entities and will usually initialize default values for the parameters.

```
package parpkg

 ! Specify parameter names and types
 parameters
  "p1":real
  "p2":integer
  "p3":string
  "p4":boolean
 end-parameters

 ! Entities for storing current parameter values
 declarations
  myp1: real
  myp2: integer
  myp3: string
  myp4: boolean
 end-declarations

 ! Set default values for parameters
 myp1:=0.25
 myp2:=10
 myp3:="default"
 myp4:=true

 !... Parameter access routines ...
end-package
```

The access routines for the four parameter types have a fixed format, namely `packagename~get[r|i|s|b]param` and `packagename~set[r|i|s|b]param` as shown in the

code extract below.

```
! Get value of a real parameter
public function parpkg~getrparam(p:string):real
 case p of
  "p1": returned:=myp1
 end-case
end-function

! Get value of an integer parameter
public function parpkg~getiparam(p:string):integer
 case p of
  "p2": returned:=myp2
 end-case
end-function

! Get value of a string parameter
public function parpkg~getsparam(p:string):string
 case p of
  "p3": returned:=myp3
 end-case
end-function

! Get value of a boolean parameter
public function parpkg~getbparam(p:string):boolean
 case p of
  "p4": returned:=myp4
 end-case
end-function

! Set value for real parameters
public procedure parpkg~setparam(p:string,v:real)
 case p of
  "p1": myp1:=v
 end-case
end-procedure

! Set value for integer parameters
public procedure parpkg~setparam(p:string,v:integer)
 case p of
  "p2": myp2:=v
 end-case
end-procedure

! Set value for string parameters
public procedure parpkg~setparam(p:string,v:string)
 case p of
  "p3": myp3:=v
 end-case
end-procedure

! Set value procedure for boolean parameters
public procedure parpkg~setparam(p:string,v:boolean)
 case p of
  "p4": myp4:=v
 end-case
end-procedure
```

A model using the package 'parpkg' will access the package parameters via Mosel's standard `getparam` and `setparam` routines (the parameter names are not case-sensitive and their names can be preceded by the package name).

```
model "Packages with parameters"
 uses 'parpkg'

 ! Display default parameter values
 writeln("Default values:",
   " p1=", getparam("parpkg.P1"), " p2=", getparam("P2"),
```

```
   " p3=", getparam("parpkg.p3"), " p4=", getparam("p4"))

 ! Change values
 setparam("p1",133)
 setparam("parpkg.p2",-77)
 setparam("P3","tluafed")
 setparam("parpkg.P4",not getparam("parpkg.P4"))

end-model
```

16.5 Namespaces

A *namespace* is a group of identifiers in a program that is distinguished by a common name (prefix). When working with mutiple packages it can be helpful to introduce namespaces in order to structure the data and to determine which model entities are accessible to other (all or preselected) packages or models.

A *fully qualified entity name* in Mosel is of the form

```
nspc~ident
```

where `nspc` is a namespace name and `ident` an identifier in the namespace. Namespaces and their access are specified via specific compiler directives at the start of the model or package. The package example `mynspkg1` below defines three namespaces ('ns1', 'ns3', and 'ns3~ns31'), two of which are restricted to a *namespace group* that comprises a second package `mynspkg2`, and the namespace 'ns3' is visible to all packages and models. The package further states via the `nssearch` directive that any unqualified entity names employed in the package should be searched for in the namespace 'ns1', meaning that the names belonging to this namespace can be used without the namespace prefix `ns1~`.

```
package mynspkg1
 namespace ns1, ns3, ns3~ns31         ! This package defines 3 namespaces:
 nsgroup ns1: "mynspkg2"              !  * ns1 + ns3~ns31 restricted to pkg2
 nsgroup ns3~ns31: "mynspkg2"         !  * ns3 is visible to all
 nssearch ns1                         ! 'ns1' can be used without prefix

 declarations
   ns3~R = 1..10
   ns1~Ar: array(ns3~R) of integer    ! Array with index set in another namespace
   vi, ns3~vi, ns3~ns31~vi: integer   ! 3 different entities
 end-declarations

 public declarations
   vp: integer                        ! This entity is visible to all
 end-declarations

 procedure ns1~proc1(val:integer)     ! Subroutine in a namespace
   ns3~vi:=val; ns3~ns31~vi:=2*val; vi:=val; vp:=val
   Ar(5):=val                         ! No prefix: 'ns1' is in search list
   writeln(" In ns1~proc1: ", vi)
 end-procedure

 public procedure proc2(val:integer)  ! Public subroutine
   writeln(" In proc2: ", val)
 end-procedure

 procedure proc3(val:integer)         ! Private subroutine
   writeln(" In proc3: ", val)
 end-procedure
end-package
```

The package `mynspkg1` shows some examples of entity and subroutine defintions for the three cases: private (`vi`, `proc3`), in a namespace (`ns3~R`, `ns1~Ar`, `ns3~vi`, `ns3~ns31~vi`, `ns1~proc1`), and

public (vp, proc2).

The second package mynspkg2 that uses functionality from mynspkg1 needs to state which namespaces are used, either via a namespace or a nssearch directive.

```
package mynspkg2
 uses 'mynspkg1'
 namespace ns3~ns31                  ! Namespace used in this package
 nssearch ns1                        ! 'ns1' can be used without prefix

 public procedure frompkg2(val: integer)
   proc1(val)                        ! Procedure in namespace 'ns1'
   writeln("  frompkg2:",ns3~ns31~vi) ! Namespace 'ns3~ns31' is not searched
   writeln("  vp=", vp)              ! Public symbol of pkg1
   writeln("  Ar(5)=", Ar(5))        ! Contained in 'ns1' (prefix optional)
 end-procedure
end-package
```

Any model or further package using the previous two packages can access the namespace 'ns3' and also define new namespaces of its own, but it is not allowed to access the other two namespaces that are restricted to this group of packages.

```
model "mynstest"
 uses 'mynspkg1', 'mynspkg2'
 namespace ns2     ! A new namespace
 nssearch ns3      ! Symbols from 'ns3' can be used without prefix

 frompkg2(5)                         ! Public routine from package mynspkg1
 writeln("n3~vi:", vi, " vp:", vp)   ! Display values of n3~vi and vp

 proc2(4)                            ! Public subroutine from mynspkg1

end-model
```

An interesting feature of namespaces is that an entire namespace can be saved via initializations to simply by indicating its name and the stored information can subsequently be used to initialize entities in some other namespace with matching names and types.

```
 declarations
   ns2~vi: integer
   I, ns2~R: range
 end-declarations

 ! Store contents of namespace 'ns3'
 initializations to "mem:mynsav"
   ns3
 end-initializations

 ! Initialize entities with matching names from the saved namespace
 initializations from "mem:mynsav"
   ns2 as "ns3"
 end-initializations
 writeln("ns2~vi:", ns2~vi)          ! Has received the value of ns3~vi

 ! Read an individual entity from the saved namespace
 initializations from "mem:mynsav"
   I as "ns3~R"
 end-initializations
 writeln("I:", I)
```

16.6 Package version management

Packages, just like Mosel models, can use the `version` compiler directive to define a version number of the format *MMM.mmm.ppp* (that is, up to 3 digits for each, major, minor, and patch numbers). Dynamic package loading relies on this number to decide whether the available package version is compatible with the required package version information recorded in a BIM file using this package. Mosel applies as a *default compatibility rule* that the (available/found) package version *ap.bp.cp* is compatible with the required package version *ar.br.cr* if *ar=ap* and *bp>=br*. For many cases, this rule is too restrictive (in particular, if you are not using any newer package functionality, a model might be expected to continue working with more recent package versions than the one it has been compiled with, as long as newer package versions continue to provide the original functionality).

Consider the following example of a package file `versionedpkg.mos` with three versions that incrementally add new (public) subroutines.

Initial version (1.0):

```
package versionedpkg
 version 1.0

!@mc.version.since 1.0.0
 public procedure showpkgver
   writeln("Using version ", getparam("model_version"),
     " of package 'versionedpkg'.")
 end-procedure

end-package
```

Second version (2.0):

```
package versionedpkg
 version 2.0
!@mc.version.compatible 1.0.0

!@mc.version.since 2.0
 public procedure myproc2
  writeln("Output from proc2")
 end-procedure

!@mc.version.since 2.0
 public procedure myoldproc
  writeln("This subroutine is deprecated with v3.0.1")
 end-procedure

!@mc.version.since 1.0.0
 public procedure showpkgver
   writeln("Using version ", getparam("model_version"),
     " of package 'versionedpkg'.")
 end-procedure

end-package
```

Current version (3.1):

```
package versionedpkg
 version 3.1
!@mc.version.compatible 1.0.0
!@mc.version.atleast 3.0

!@mc.version.since 2.0
 public procedure myproc2
  writeln("Output from proc2")
 end-procedure
```

```
!@mc.version.since 3.0.2
 public procedure myproc3
  writeln("Output from proc3")
 end-procedure

!@mc.version.deprecated 3.0.1
!@mc.version.since 2.0
 public procedure myoldproc
  writeln("This subroutine is deprecated with v3.0.1")
 end-procedure

! Unversioned subroutine (defaults to current package version)
 public procedure mynewproc
  writeln("Output from newproc")
 end-procedure

!@mc.version.since 1.0.0
 public procedure showpkgver
   writeln("Using version ", getparam("model_version"),
     " of package 'versionedpkg'.")
 end-procedure

end-package
```

These three versions of the package 'versionedpkg' specify *version management annotations* to define version compatibility rules and deprecation markup for subroutines. (The concept of annotations is explained more in detailed in Chapter 18.) A Mosel program using package functionality might look as follows:

```
model "test package versioning"
 uses "versionedpkg"

 ! Calling a subroutine defined since the initial package version
 showpkgver

end-model
```

If we have used the initial package version (1.0.0) for compiling this model we shall see in the output of Mosel's `exam` command on the resulting BIM file `versionedpkg_test.bim` the following dependency information:

```
Model  `test package versioning' version 0.0.0
 file: .\versionedpkg_test.bim
 sys. com.:  `versionedpkg_test.mos',mc6.4.0
(...)
 pkg. req.: versionedpkg (1.0.0)

Procedures and Functions:
 procedure showpkgver [reqmt]
```

This means, that the compiled model can be run with a BIM file generated from any of the package versions above, since all of them are marked as *compatible* with version 1.0.0. The actual package version that is in use when running the model file will be displayed by the routine 'showpkgver', for example, with the third package version we will see the output:

```
Using version 3.1.0 of package 'versionedpkg'.
```

If we now recompile the model file with a BIM file generated with the third version (3.1.0), we see the *required package version* in the output from the `exam` command change to 3.0.0 (we are not using any functionality from this version, but this package is tagged as requiring *at least* version 3.0). If we now add a call to the subroutine 'myproc3' into the model, the minimum required version will be raised to 3.0.2. Using the subroutine 'mynewproc' that does not have any version annotation will further raise the

minimum version requirement to the current package version (3.1.0).

If we call the subroutine 'myoldproc' along with 'myproc3' from our model file, we shall see a deprecation warning displayed since this routine is marked as deprecated since 3.0.1, which is in conflict with the minimum version 3.0.2 that is induced by the use of 'myproc3':

```
Mosel: W-241 at line 39 of  `versionedpkg_test.mos': Using deprecated subroutine  `myoldproc'.
```

16.7 Packages vs. modules

The possibility of writing packages introduces a second form of libraries for Mosel, the first being *modules* (see the 'Mosel Native Interface User Guide' for further detail). The following list summarizes the main differences between packages and modules.

- **Definition**
  - *Package*
    - library written in the Mosel language
  - *Module*
    - dynamic library written in C that obeys the conventions of the Mosel Native Interface
- **Functionality**
  - *Package*
    - define
      - symbols
      - subroutines
      - types
      - control parameters
  - *Module*
    - extend the Mosel language with
      - constant symbols
      - subroutines
      - operators
      - types
      - control parameters
      - I/O drivers
- **Efficiency**
  - *Package*
    - like standard Mosel models
  - *Module*
    - faster execution speed
    - higher development effort
- **Use**
  - *Package*
    - making parts of Mosel models re-usable

    - deployment of Mosel code whilst protecting your intellectual property
  - *Module*
    - connection to external software
    - time-critical tasks
    - definition of new I/O drivers and operators for the Mosel language

As can be seen from the list above, the choice between packages and modules depends largely on the contents and intended use of the library you wish to write.

CHAPTER 17

Language extensions

It has been said before that the functionality of the Mosel language can be extended by means of *modules*, that is, dynamic libraries written in C/C++. All through this manual we have used the module *mmxprs* to access Xpress Optimizer. Other modules we have used are *mmsheet* and *mmodbc* (access to spreadsheets and databases, see Section 2.2.5), and *mmsystem* (Sections 5.1 and 11.1).

The full distribution of Mosel includes other functionality (modules and I/O drivers) that has not yet been mentioned in this document. In the following sections we give an overview with links where to find additional information.

17.1 Generalized file handling

The notion of *(data) file* encountered repeatedly in this user guide seems to imply a physical file. However, Mosel language statements (such as `initializations from` / `to`, `fopen` and `fclose`, `exportprob`) and the Mosel library functions (*e.g.*, `XPRMcompmod`, `XPRMloadmod`, or `XPRMrunmod`) actually work with a much more general definition of a *file*, including (but not limited to)

- a physical file (text or binary)
- a block of memory
- a file descriptor provided by the operating system
- a function (callback)
- a database

The type of the file is indicated by adding to its name the name of the *I/O driver* that is to be used to access it. In Section 2.2.5 we have used `mmodbc.odbc:blend.mdb` to access an MS Access database via the ODBC driver provided by the module *mmodbc*. If we want to work with a file held in memory we may write, for instance, `mem:filename`. The default driver (no driver prefix) is the standard Mosel file handling.

More generally, an extended file name has the form *driver_name:file_name* or *module_name.driver_name:file_name* if the driver is provided by the module *module_name*. The structure of the *file_name* part of the extended file name is specific to the driver, it may also consist of yet another extended file name (*e.g.* `zlib.gzip:tmp:myfile.txt`).

17.1.1 Displaying the available I/O drivers

The Mosel core drivers can be displayed from the command line with the following command (the listing will also include any drivers that are provided by currently loaded modules):

```
mosel exam -i
```

The drivers provided by modules are displayed by the `exam` command for the corresponding module (in this example: *mmodbc*)

```
mosel exam -i mmodbc
```

Library drivers (in particular the Java module *mmjava* that is embedded in the Mosel core and also the *mmdotnet* module on Windows platforms) can be displayed with the help of the corresponding program `mmdispdso.[c|cs|java]` in the subdirectory `examples/mosel/Library` of the Xpress distribution. The command for running the Java version might look as follows (please refer to the provided `makefile`):

```
java -classpath $XPRESSDIR/xprm.jar:. mmdispdso mmjava
```

17.1.2 List of I/O drivers

The standard distribution of Mosel defines the following I/O drivers (`tee`, `null`, `bin`, and `tmp` are documented in the 'Mosel Language Reference Manual'; the drivers `sysfd`, `mem`, `cb`, and `raw` that mainly serve when interfacing Mosel with a host application are documented in the 'Mosel Libraries Reference Manual'):

`bin` Write (and read) data files in a platform independent binary format. The `bin` driver can only be used for `initializations` blocks as a replacement of the default driver. Files in `bin` format are generally smaller than the ASCII equivalent and preserve accuracy of floating point numbers.
Example: the following outputs data to a text file in binary format

```
initializations to "bin:results.txt"
```

Another likely use of `bin` is in combination with `mem` or `shmem` for exchanging data in memory (see Section 17.2.3):

```
initializations to "bin:shmem:results"
```

`cb` Use a (callback) function as a file (*e.g.*, for reading and writing dynamically sized data in `initializations` blocks, see the examples in Section 13.4.3, or to write your own output or error stream handling functions when working with the Mosel libraries, see Section 13.5 for an example).

`mem` Use memory instead of physical files for reading or writing data (*e.g.*, for exchanging data between a model and the calling C application as shown in Section 13.4 or for compiling/loading a model to/from memory when working with the Mosel libraries).
Example: the following lines will compile the Mosel model `burglar2.mos` to memory and then load it from memory (full example in file `ugcompmem.c`).

```
XPRMmodel mod;
char bimfile[2000];              /* Buffer to store BIM file */
char bimfile_name[64];           /* File name of BIM file */

XPRMinit())                      /* Initialize Mosel */

/* Prepare file name for compilation using 'mem' driver: */
/*   "mem:base_address/size[/actual_size_of_pointer]"    */
sprintf(bimfile_name, "mem:%p/%d", bimfile, (int)sizeof(bimfile));

                                 /* Compile model file to memory */
XPRMcompmod(NULL, "burglar2.mos", bimfile_name, "Knapsack example"))

                                 /* Load a BIM file from memory */
mod = XPRMloadmod(bimfile_name, NULL);
```

`null` Disable a stream.
Example: adding the line

```
fopen("null:", F_OUTPUT)
```

in a Mosel model will disable all subsequent output by this model (until the output stream is closed or a new output stream is opened).

`raw` Implementation of the `initializations` block in binary mode, typically used in combination with `mem` for data exchange with a C host application (see Section 13.4).

`sysfd` Working with operating system file descriptors (for instance, file descriptors returned by the C function `open`).
Example: in a C program, the line

```
XPRMsetdefstream(NULL, XPRM_F_ERROR, "sysfd:1");
```

will redirect the Mosel error stream to the default output stream.

`tee` Output into up to 6 files simultaneously (*e.g.*, to display a log on screen and write it to a file at the same time).
Example: adding the line

```
fopen("tee:result.txt&tmp:log.txt&", F_OUTPUT)
```

in a Mosel model will redirect all subsequent model output to the files `result.txt` and `tmp:log.txt`, and at the same time display the output on the default output (screen), the latter is denoted by the `&` sign at the end of the filename string. The output to both locations is terminated by the statement

```
fclose(F_OUTPUT)
```

after which output will again only go to default output.

`tmp` Extension to the default driver that locates the specified file in the temporary directory used by Mosel.
Example: adding the line

```
fopen("tmp:log.txt", F_OUTPUT+F_APPEND)
```

redirects all subsequent output from a models to the file `log.txt` that is located in Mosel's temporary directory. It is equivalent to

```
fopen(getparam("TMPDIR") + "/log.txt", F_OUTPUT+F_APPEND)
```

Some modules, listed below in alphabetical order, define additional I/O drivers. The drivers are documented with the corresponding module in the 'Mosel Language Reference Manual':

- **mmdotnet**

  `dotnet` Use a C# stream or object in place of a file in Mosel.
  Example: The following C# program extract uses the `dotnet` driver to send data in standard initializations text format via a stream from the C# host program to a model (the model file `burglar13.mos` is the same as in Section 13.4.3, it uses the parameter *DATAFILE* as the filename for an `initializations from` block that expects to read data with the label *'DATA'*.)

```
// String containing initialization data for the model
const string BurglarDat =
  "DATA: [(\"camera\") [15 2] (\"necklace\") [100 20] " +
  "(\"vase\") [90 20] (\"picture\") [60 30] (\"tv\") [40 40] " +
  "(\"video\") [15 30] (\"chest\") [10 60] (\"brick\") [1 10] ]\n";

static void Main(string[] args) {
```

```
  // Initialize Mosel
  XPRM mosel = XPRM.Init();

  // Compile and load the Mosel model
  XPRMModel model = mosel.CompileAndLoad("burglar13.mos");

  // Bind a stream based on the BurglarDat data to the name 'BurglarIni'
  // where the model will expect to find its initialization data
  model.Bind("BurglarIni", new StringReader(BurglarDat));

  // Pass data location as a parameter to the model
  model.SetExecParam("DATAFILE","dotnet:BurglarIni");

  // Run the model
  model.Run();
}
```

`dotnetraw` Exchange of data between a Mosel model and the C# application running the model using C# array structures; C# version of `raw`.
Example: Send the data held in the two arrays `vdata` and `wdata` to a Mosel and retrieve solution data into the array `solution`. We use the option *noindex* with the *dotnetraw* driver to indicate that all data are saved in dense format (*i.e.* as arrays containing just the data values, without any information about the indices).

```
// Arrays containing initialization data for the model
static double[] vdata = new double[] {15,100,90,60,40,15,10, 1};
static double[] wdata = new double[] { 2, 20,20,30,40,30,60,10};

// Main entry point for the application
static void Main(string[] args) {
  // Initialize Mosel
  XPRM mosel = XPRM.Init();

  // Compile and load the Mosel model
  XPRMModel model = mosel.CompileAndLoad("burglar8d.mos");

  // Associate the .NET objects with names in Mosel
  model.Bind("vdat", vdata);
  model.Bind("wdat", wdata);

  // Create a new array for solution data and bind it to the name 'sol'
  double[] solution = new double[8];
  mosel.Bind("sol", solution);

  // Pass data location as a parameter to the model
  model.ExecParams =
    "VDATA='noindex,vdat',WDATA='noindex,wdat',SOL='noindex,sol'";

  // Run the model
  model.Run();

  // Print the solution
  Console.WriteLine("Objective value: {0}", model.ObjectiveValue);
  for (int i=0;i<8;i++)
    Console.Write(" take({0}): {1}", (i+1), solution[i]);
  Console.WriteLine();
}
```

The model file `burglar8d.mos` uses the driver name as the file name in the `initializations` sections:

```
 initializations from 'dotnetraw:'
  VALUE as VDATA  WEIGHT as WDATA
 end-initializations
...

 initializations to 'dotnetraw:'
  soltake as SOL
```

```
end-initializations
```

- **mmetc**

  `diskdata` Access data in text files in diskdata format (see Sections 3.4.3 and 10.2.3).

- **mmhttp**

  `url` Access files that are stored on an HTTP enabled file serverfor reading, writing, or deletion (via `fdelete`).
  Example 1: the following command downloads and executes the Mosel BIM file `mymodel.bim` that is stored on the web server `myserver`:

```
mosel run mmhttp.url:http://myserver/mymodel.bim
```

  Example 2: the following lines of Mosel code save data held in the model object `results` to an XML format file on the server `myserver` that needs to be able to accept HTTP PUT requests.

```
uses "mmxml"
declarations
  results: xmldoc
end-declarations
save(results, "mmhttp.url:http://myserver/myresults.xml")
```

- **mmjava**

  `java` Use a Java stream or a `ByteBuffer` in place of a file in Mosel (*e.g.* for redirecting default Mosel streams to Java objects, see the example in Section 14.1.8).
  Example 1: in a Java program, the line

```
mosel.setDefaultStream(XPRM.F_ERROR, "java:java.lang.System.out");
```

  (where `mosel` is an object of class `XPRM`) will redirect the Mosel error stream to the default output stream of Java.
  Example 2: the following lines will compile the Mosel model `burglar2.mos` to memory and then load it from memory (full example in the file `ugcompmem.java`).

```
XPRM mosel;
XPRMModel mod;
ByteBuffer bimfile;                  // Buffer to store BIM file

mosel = new XPRM();                  // Initialize Mosel

                              // Prepare file names for compilation:
bimfile=ByteBuffer.allocateDirect(2048);  // Create 2K byte buffer
mosel.bind("mybim", bimfile);        // Associate Java obj. with a
                                     // Mosel name

                                     // Compile model to memory
mosel.compile("", "burglar2.mos", "java:mybim", "");

bimfile.limit(bimfile.position());   // Mark end of data in buffer
bimfile.rewind();                    // Back to the beginning

mod=mosel.loadModel("java:mybim");   // Load BIM file from memory
mosel.unbind("mybim");               // Release memory
bimfile=null;
```

  `jraw` Exchange of data between a Mosel model and the Java application running the model; Java version of `raw`. See Section 14.1.7 for examples.

- **mmjobs**

`shmem` Use shared memory instead of physical files for reading or writing data (*e.g.*, for exchanging data between several models executed concurrently—one model writing, several models reading—as shown in Section 17.2.3, or for compiling/loading a model to/from memory from within another model, see Section 17.2.2).

`mempipe` Use memory pipes for reading or writing data (*e.g.*, for exchanging data between several models executed concurrently—one model reading, several models writing; see Section 'Dantzig-Wolfe decomposition' of the whitepaper 'Multiple models and parallel solving with Mosel' for an example).

`rcmd` Starts the specified command in a new process and connects its standard input and output streams to the calling Mosel instance.
Example: The following line starts a Mosel instance on a remote computer with the name `some_other_machine` connecting to it via *rsh*. The command `mosel -r` starts Mosel in remote mode.

```
rcmd:rsh some_other_machine mosel -r
```

`rmt` Can be used with any routine expecting a physical file for accessing files on remote instances. A particular instance can be specified by prefixing the file name by its node number enclosed in square brackets. See the distributed computing versions of the models in the whitepaper *Multiple models and parallel solving with Mosel* for further examples.
Example: the following loads the BIM file `mymodel.bim` that is located at the parent node (-1) of the current instance into the model `myModel` of the Mosel instance `myInst`.

```
load(myInst, myModel, "rmt:[-1]mymodel.bim")
```

The `rmt` driver can be combined with `cb`, `sysfd`, `tmp` or `java`, such as

```
fopen("rmt:tmp:log.txt", F_OUTPUT)
```

`xsrv` Connects to the specified host running the Mosel Remote Launcher `xprmsrv`. Optionally, the port number, a context name, a password, and environment variable settings can be specified.
Example: the following starts a new Mosel instance (possibly using a different Xpress version) on the current machine, redefining Mosel's current working directory and the environment variable `MYDATA`.

```
xsrv:localhost/myxpress|MOSEL_CWD=C:\workdir|MYDATA=${MOSEL_CWD}\data
```

`xssh` Secure version of the *xsrv* driver to connect to the specified host running the Mosel Remote Launcher `xprmsrv` through a secure SSH tunnel.

- **mmoci**

`oci` Access an Oracle database for reading and writing in `initializations` blocks (see the whitepaper *Using ODBC and other database interfaces with Mosel* for further examples).
Example: the OCI driver is used with a connection string that contains the database name, user name and password in the following format

```
initializations from "mmoci.oci:myusername/mypassword@dbname
```

- **mmodbc**

  `odbc` Access data in external data sources via an ODBC connection (see Section 2.2.5 for an example).

- **mmsheet**

  `csv` Access spreadsheets in CSV format. In addition to the standard options supported by other Mosel spreadsheet drivers (such as `grow` for dynamic sizing of output ranges and `noindex` for dense data format), this driver can be configured with field and decimal separators and also with the values representing 'true' and 'false'.
  Example: the following Mosel code for reading data from the file `mydata.csv` sets the separator sign to ';', a comma is used as decimal separator, and the Boolean values true and false are represented by 'y' and 'n' respectively:

```
initializations from "mmsheet.csv:dsep=,;fsep=;;true=y;false=n;mydata.csv
  A as "[B2:D8]"
  B as "[E2:Z10](#3,#1)"       ! 3rd and 1st column from the range
end-initializations
```

  Notice that with CVS format files there is no notion of range or field names and the cell positions need to be used to specify the data location.

  `excel` Access data in MS Excel spreadsheets directly (see the example in Section 2.2.5.1).

  `xls` Access spreadsheets in Excel's XLS format (see Section 2.2.5.3 for an example).

  `xlsx` Access spreadsheets in Excel's XLSX and XLSM formats (usage in analogy to the XLS example shown in Section 2.2.5.3).

- **mmsystem**

  `pipe` Open a pipe and start an external program which is used as input or output stream for a Mosel model.
  Example: the following will start gnuplot and draw a 3-dimensional sphere with data for the radius R and position (X,Y,Z) defined in the Mosel model:

```
fopen("mmsystem.pipe:gnuplot -p", F_OUTPUT+F_LINBUF)
writeln('set parametric')
writeln('set urange [0:2*pi]')
writeln('set vrange [0:2*pi]')
writeln('set pm3d depthorder hidden3d 3')
writeln("splot cos(u) * cos(v) *", R, "+", X,
        ", sin(u) * cos(v)*", R, "+", Y,
        ", sin(v)*", R, "+", Z, " notitle with pm3d")
writeln("pause mouse keypress,close")
```

  `text` Use a (multiline) text as a file (see Section 17.6 for further detail on the type 'text').
  Example: the following will compile the model source held in the text `source_of_model` to a BIM file in a temporary directory (file `ugcompfrommem.mos`):

```
 public declarations
  source_of_model= `SUBMODELSOURCE
 model Burglar
  uses 'mmxprs'

  declarations
   WTMAX = 102                    ! Maximum weight allowed
   ITEMS = 1..8                   ! Index range for items
   VALUE: array(ITEMS) of real    ! Value of items
```

```
  WEIGHT: array(ITEMS) of real   ! Weight of items
  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

 VALUE :: [15, 100, 90, 60, 40, 15, 10,  1]
 WEIGHT:: [ 2,  20, 20, 30, 40, 30, 60, 10]

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX
! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the problem
 writeln("Solution:\n Objective: ", getobjval)
end-model
SUBMODELSOURCE `
end-declarations

! Compile the model from memory
if compile("", "text:source_of_model", "tmp:burglar.bim")<>0 then
  exit(1)
end-if
```

- **zlib**

`deflate` Handles files compressed using the *zlib* compression format.
Example: decompress the file `myfile.gz` to `myfile`:

```
fcopy("zlib.deflate:myfile.gz", "myfile")
```

`gzip` Handles files compressed using the *gzip* compression format.
Example: the following statement creates the compressed file `myfile.gz` from `myfile.txt`:

```
fcopy("myfile.txt", "zlib.gzip:myfile.gz")
```

The reader is referred to the whitepaper *Generalized file handling in Mosel* that is provided as a part of the Xpress documentation in the standard distribution and also on the Xpress website under 'Product Documentation' for further explanation of this topic and a documented set of examples, including some user-written I/O drivers.

17.2 Multiple models and parallel solving with *mmjobs*

The module *mmjobs* makes it possible to exchange information between models running concurrently—locally or in a network. Its functionality includes facilities for handling Mosel instances (*e.g.* connecting and disconnecting Mosel instances, access to remote files), model management (*e.g.* compiling, running, or interrupting a model from within a second model), synchronization of concurrent models based on event queues, and a shared memory I/O driver for an efficient exchange of data between models that are executed concurrently.

Several complete examples (including examples of Benders decomposition and Dantzig-Wolfe decomposition) of the use of module *mmjobs* are described in the whitepaper *Multiple models and parallel solving with Mosel* that is provided as a part of the Xpress documentation and also on the 'Product Documentation' page of the Xpress website. We show here how to use the basic functionality for executing a model from a second model.

17.2.1 Running a model from another model

As a test case, we shall once more work with model `prime.mos` from Section 8.3. In the first instance, we now show how to compile and run this model from a second model, `runprime.mos`:

```
model "Run model prime"
 uses "mmjobs"

 declarations
  modPrime: Model
  event: Event
 end-declarations
                              ! Compile 'prime.mos'
 if compile("prime.mos")<>0: exit(1)

 load(modPrime, "prime.bim")   ! Load bim file

 run(modPrime, "LIMIT=50000")  ! Start execution and
 wait(2)                       ! wait 2 seconds for an event

 if isqueueempty then          ! No event has been sent...
  writeln("Model too slow: stopping it!")
  stop(modPrime)               ! ... stop the model,
  wait                         ! ... and wait for the termination event
 end-if
                              ! An event is available: model finished
 event:=getnextevent
 writeln("Exit status: ", getvalue(event))
 writeln("Exit code  : ", getexitcode(modPrime))

 unload(modPrime)             ! Unload the submodel
end-model
```

The `compile` command generates the BIM file for the given submodel; the command `load` loads the binary file into Mosel; and finally we start the model with the command `run`. The `run` command is not used in its basic version (single argument with the model reference): here its second argument sets a new value for the parameter `LIMIT` of the submodel.

In addition to the standard compile–load–run sequence, the model above shows some basic features of interaction with the submodel: if the submodel has not terminated after 2 seconds (that is, if it has not sent a termination message) it is stopped by the controlling (parent) model. After termination of the submodel (either by finishing its calculations within less than 2 seconds or stopped by the parent model) its termination status and the exit value are retrieved (functions `getvalue` and `getexitcode`). Unloading a submodel explicitly as shown here is only really necessary in larger applications that continue after the termination of the submodel, so as to free the memory used by it.

Note: our example model shows an important property of submodels—they are running in parallel to the parent model and also to any other submodels that may have been started from the parent model. It is therefore essential to insert `wait` at appropriate places to coordinate the execution of the different models.

17.2.2 Compiling to memory

The model shown in the previous section compiles and runs a submodel. The default compilation of a Mosel file `filename.mos` generates a binary model file `filename.bim`. To avoid the generation of physical BIM files for submodels we may compile the submodel to memory, making use of the concept of I/O drivers introduced in Section 17.1.

Compiling a submodel to memory is done by replacing the standard `compile` and `load` commands by the following lines (model `runprime2.mos`):

```
if compile("","prime.mos","shmem:bim")<>0: exit(1)

load(modPrime,"shmem:bim")    ! Load bim file from memory...
fdelete("shmem:bim")          ! ... and release the memory block
```

The full version of `compile` takes three arguments: the compilation flags (*e.g.*, use `"g"` for debugging), the model file name, and the output file name (here a label prefixed by the name of the shared memory driver). Having loaded the model we may free the memory used by the compiled model with a call to `fdelete` (this subroutine is provided by the module *mmsystem*).

17.2.3 Exchanging data between models

When working with submodels we are usually not just interested in executing the submodels, we also wish to retrieve their results in the main model. This is done most efficiently by exchanging data in (shared) memory as shown in the model `runprimeio.mos` below. Besides the retrieval and printout of the solution we have replaced the call to `stop` by sending the user event 'STOPMOD' to the submodel: instead of simply terminating the submodel this event will make it interrupt its calculations and write out the current solution. To make sure that the submodel is actually running at the point where we sent the 'STOPMOD' event, we have also introduced an event sent from the submodel to its parent model to indicate the point of time when it starts the calculations (with heavy operating system loads the actual submodel start may be delayed). Once the submodel has terminated (after sending the 'STOPMOD' event we `wait` for the model's termination message) we may read its solution from memory, using the `initializations` block with the drivers `raw` (binary format) and `shmem` (read from shared memory).

```
model "Run model primeio"
 uses "mmjobs"

 declarations
  modPrime: Model
  NumP: integer                ! Number of prime numbers found
  SetP: set of integer         ! Set of prime numbers
  STOPMOD = 2                  ! "Stop submodel" user event
  MODREADY = 3                 ! "Submodel ready" user event
 end-declarations

                               ! Compile 'prime.mos'
 if compile("primeio.mos")<>0: exit(1)

 load(modPrime, "primeio.bim") ! Load bim file

                               ! Disable model output
 setdefstream(modPrime,"","null:","null:")
 run(modPrime, "LIMIT=35000")  ! Start execution and
 wait                          ! ... wait for an event
 if getclass(getnextevent) <> MODREADY then
  writeln("Problem with submodel run")
  exit(1)
 end-if

 wait(2)                       ! Let the submodel run for 2 seconds

 if isqueueempty then          ! No event has been sent...
  writeln("Model too slow: stopping it!")
  send(modPrime, STOPMOD, 0)   ! ... stop the model, then wait
  wait
 end-if
 dropnextevent                 ! Discard end events

 initializations from "bin:shmem:resdata"
  NumP  SetP as "SPrime"
 end-initializations

 writeln(SetP)                 ! Output the result
```

```
 writeln(" (", NumP, " prime numbers.)")

 unload(modPrime)
end-model
```

We now have to modify the submodel (file `primeio.mos`) correspondingly: at its start it sends the 'MODREADY' event to trigger the start of the time measurement in the parent model and it further needs to intercept the 'STOPMOD' event interrupting the calculations (via an additional test `isqueueempty` for the `repeat-until` loop) and write out the solution to memory in the end:

```
model "Prime IO"
 uses "mmjobs"

 parameters
  LIMIT=100                     ! Search for prime numbers in 2..LIMIT
  OUTPUTFILE = "bin:shmem:resdata"    ! Location for output data
 end-parameters

 declarations
  SNumbers: set of integer      ! Set of numbers to be checked
  SPrime: set of integer        ! Set of prime numbers
  MODREADY = 3                  ! "Submodel ready" user event
 end-declarations

 send(MODREADY,0)               ! Send "model ready" event

 SNumbers:={2..LIMIT}

 writeln("Prime numbers between 2 and ", LIMIT, ":")

 n:=2
 repeat
   while (not(n in SNumbers)) n+=1
   SPrime += {n}                ! n is a prime number
   i:=n
   while (i<=LIMIT) do          ! Remove n and all its multiples
     SNumbers-= {i}
     i+=n
   end-do
 until (SNumbers={} or not isqueueempty)

 NumP:= getsize(SPrime)

 initializations to OUTPUTFILE
  NumP  SPrime
 end-initializations

end-model
```

Note: since the condition `isqueueempty` is tested only once per iteration of the `repeat-until` loop, the termination of the submodel is not immediate for large values of `LIMIT`. If you wish to run this model with very large values, please see Section 15.2 for an improved implementation of the prime number algorithm that considerably reduces its execution time.

17.2.4 Distributed computing

The module *mmjobs* not only allows the user to start several models in parallel on a given machine, it also makes it possible to execute models remotely and to coordinate their processing. With only few additions, the model from Section 17.2.1 is extended to form model version `runprimedistr.mos` that launches the submodel `prime.mos` on another Mosel instance (either on the local machine as in the present example or on some other machine within a network specified by its name or IP address in the `connect` statement): we need to create a new Mosel instance (through a call to `connect`) and add an additional argument to the `load` statement to specify the Mosel instance we wish to use. All else

remains the same as in the single-instance version.

```
model "Run model prime remotely"
 uses "mmjobs"

 declarations
  moselInst: Mosel
  modPrime: Model
  event: Event
 end-declarations
                               ! Compile 'prime.mos' locally
 if compile("prime.mos")<>0: exit(1)

                               ! Start a remote Mosel instance:
                               ! "" means the node running this model
 if connect(moselInst, "")<>0: exit(2)

                               ! Load bim file into remote instance
 load(moselInst, modPrime, "rmt:prime.bim")

 run(modPrime, "LIMIT=50000")  ! Start execution and
 wait(2)                       ! wait 2 seconds for an event

 if isqueueempty then          ! No event has been sent...
  writeln("Model too slow: stopping it!")
  stop(modPrime)               ! ... stop the model, then wait
  wait
 end-if
                               ! An event is available: model finished
 event:=getnextevent
 writeln("Exit status: ", getvalue(event))
 writeln("Exit code  : ", getexitcode(modPrime))

 unload(modPrime)              ! Unload the submodel
end-model
```

This model can be extended to include data exchange between the parent and the submodel exactly in the same way as in the example of Section 17.2.3. The main difference (besides the connection to a remote instance) lies in the use of the driver *rmt* to denote the Mosel instance where data is to be saved. In our case, we wish to save data on the instance running the parent model, meaning that we need to use the `rmt:` prefix when writing output within the submodel. The new output file name is passed into the submodel via the runtime parameter `OUTPUTFILE`:

```
 run(modPrime, "LIMIT=35000,OUTPUTFILE=bin:rmt:shmem:resdata")
```

The parent model then simply reads as before from its own instance:

```
 initializations from "bin:shmem:resdata"
  NumP  SetP as "SPrime"
 end-initializations
```

The Mosel model for the extended version including data exchange is provided in the file `runprimeiodistr.mos`.

17.3 Graphics and GUIs

Different components of FICO Xpress Optimization provide graphics and GUI functionality for Mosel models:

- Users may enrich their Mosel models with graphical output using the module *mmsvg*.
- **Xpress Insight** embeds Mosel models into a multi-user application for deploying optimization

models in a distributed client-server architecture. Through the Xpress Insight GUI, business users interact with Mosel models to evaluate different scenarios and model configurations without directly accessing to the model itself.

- XML is a widely used data format, particularly in the context of web-based applications. The Mosel module *mmxml* provides functionality for generating and handling XML documents. *mmxml* can also be used to produce HTML format output from Mosel that can be incorporated into Xpress Insight applications.

The functionality of modules *mmxml* and *mmsvg* is documented in the Mosel Language Reference Manual. Xpress Insight has several manuals and guides for developers and GUI users, most importantly the 'Xpress Insight Developer Guide' and 'Xpress Insight Web Client User Guide'; the corresponding examples are located in the subdirectory `examples/insight` of the Xpress distribution.

17.3.1 Drawing user graphs with mmsvg

The graphic in Figure 17.1 is an example of using *mmsvg* to produce a graphical representation of the solution to the transport problem from Section 3.2.

It was obtained by calling the following procedure `draw_solution` at the end of the model file (that is, after the call to `minimize`).

```
procedure draw_solution
 declarations
  YP: array(PLANT) of integer           ! y-coordinates of plants
  YR: array(REGION) of integer          ! y-coordinates of sales regions
 end-declarations

 ! Scale the size of the displayed graph
 svgsetgraphviewbox(0,0,4,getsize(REGION)+1)
 svgsetgraphscale(100)

 ! Determine y-coordinates for plants and regions
 ct:= 1+floor((getsize(REGION)-getsize(PLANT))/2)
 forall(p in PLANT, ct as counter) YP(p):= ct

 ct:=1
 forall(r in REGION, ct as counter) YR(r):= ct

 ! Draw the plants
 svgaddgroup("PGr", "Plants", svgcolor(0,63,95))
 forall(p in PLANT) svgaddtext(0.55, YP(p)-0.1, p)

 ! Draw the sales regions
 svgaddgroup("RGr", "Regions", svgcolor(0,157,169))
 forall(r in REGION) svgaddtext(3.1, YR(r)-0.1, r)

 ! Draw all transport routes
 svgaddgroup("TGr", "Routes", SVG_GREY)
 forall(p in PLANT, r in REGION | exists(TRANSCAP(p,r)) )
  svgaddline(1, YP(p), 3, YR(r))

 ! Draw the routes used by the solution
 svgaddgroup("SGr", "Solution", SVG_ORANGE)
 forall(p in PLANT, r in REGION | exists(flow(p,r)) and getsol(flow(p,r)) > 0)
  svgaddarrow(1, YP(p), 3, YR(r))

 ! Save graphic in SVG format
 svgsave("transport.svg")

 ! Display the graphic
 svgrefresh
 svgwaitclose
end-procedure
```

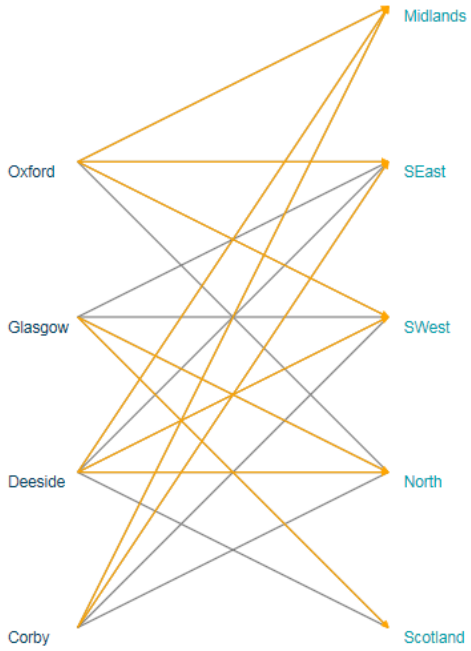


Figure 17.1: User graph for the transport problem

17.3.2 XML and HTML

HTML files are simple text files—their contents can be generated as free-format output from Mosel (see for example Section 10.2). However, more elegantly we can use Mosel's XML module *mmxml* to generate HTML documents.

17.3.2.1 mmxml

The module *mmxml* provides an XML parser and generator for the manipulation of XML documents from Mosel models. An *XML document* is stored as a list of nodes. *mmxml* supports the node types 'element', 'text', 'comment', CDATA, 'processing instruction' and DATA (see section *mmxml* of the Mosel Language Reference Manual for further detail). Each node is characterized by a *name* and a *value*. Element nodes have also an ordered list of child nodes. The *root* node is a special element node with no name, no parent and no successor that includes the entire document as its children.

The type `xmldoc` represents an XML document stored in the form of a tree. Each node of the tree is identified by a *node number* (an integer) that is attached to the document (*i.e.* a node number cannot be shared by different documents and in two different documents the same number represents two different nodes). The *root* node of the document has number `0`. Nodes can be retrieved using a *path* similar to a directory path used to locate a file (usually called *XML path*).

17.3.2.2 Reading and writing XML data

Data for the 'Transport' problem has so far been given as a text data file in `initializations` format. We now wish to read in the same data from the XML file `transprt.xml` shown here:

```
<transport fuelcost="17">
  <demand>
    <region name="Scotland">2840</region>
    <region name="North">2800</region>
    <region name="SWest">2600</region>
    <region name="SEast">2820</region>
```

```
    <region name="Midlands">2750</region>
  </demand>
  <plantdata>
    <plant name="Corby">
      <capacity>3000</capacity>
      <cost>1700</cost>
    </plant>
    <plant name="Deeside">
      <capacity>2700</capacity>
      <cost>1600</cost>
    </plant>
    <plant name="Glasgow">
      <capacity>4500</capacity>
      <cost>2000</cost>
    </plant>
    <plant name="Oxford">
      <capacity>4000</capacity>
      <cost>2100</cost>
    </plant>
  </plantdata>
  <routes>
    <route from="Corby" to="North" capacity="1000" distance="400"/>
    <route from="Corby" to="SWest" capacity="1000" distance="400"/>
    <route from="Corby" to="SEast" capacity="1000" distance="300"/>
    <route from="Corby" to="Midlands" capacity="2000" distance="100"/>
    <route from="Deeside" to="Scotland" capacity="1000" distance="500"/>
    <route from="Deeside" to="North" capacity="2000" distance="200"/>
    <route from="Deeside" to="SWest" capacity="1000" distance="200"/>
    <route from="Deeside" to="SEast" capacity="1000" distance="200"/>
    <route from="Deeside" to="Midlands" capacity="300" distance="400"/>
    <route from="Glasgow" to="Scotland" capacity="3000" distance="200"/>
    <route from="Glasgow" to="North" capacity="2000" distance="400"/>
    <route from="Glasgow" to="SWest" capacity="1000" distance="500"/>
    <route from="Glasgow" to="SEast" capacity="200" distance="900"/>
    <route from="Oxford" to="North" capacity="2000" distance="600"/>
    <route from="Oxford" to="SWest" capacity="2000" distance="300"/>
    <route from="Oxford" to="SEast" capacity="2000" distance="200"/>
    <route from="Oxford" to="Midlands" capacity="500" distance="400"/>
  </routes>
</transport>
```

The Mosel code (file `transport_xml.mos`) for reading this XML file first loads the entire document (`load`). We then need to retrieve the nodes we are interested in from the XML document tree structure. This is achieved by selecting lists of nodes that satisfy some specified condition (here: a specific XML path that describes the location of the desired nodes in the document, such as `transport/demand/region`). XML documents are saved as `text`, we therefore use functions like `getrealvalue` or `getstrattr` to retrieve data and indices of the desired type into the model data structures.

```
uses "mmxml"

declarations
 REGION: set of string                ! Set of customer regions
 PLANT: set of string                 ! Set of plants

 DEMAND: array(REGION) of real        ! Demand at regions
 PLANTCAP: array(PLANT) of real       ! Production capacity at plants
 PLANTCOST: array(PLANT) of real      ! Unit production cost at plants
 TRANSCAP: dynamic array(PLANT,REGION) of real
                                      ! Capacity on each route plant->region
 DISTANCE: dynamic array(PLANT,REGION) of real
                                      ! Distance of each route plant->region
 FUELCOST: real                       ! Fuel cost per unit distance

 AllData: xmldoc                      ! XML document
 NodeList: list of integer            ! List of XML nodes
end-declarations
```

```
load(AllData, "transprt.xml")          ! Load the entire XML document

getnodes(AllData, "transport/demand/region", NodeList)
forall(l in NodeList)                  ! Read demand data
  DEMAND(getstrattr(AllData,l,"name")):= getrealvalue(AllData, l)

getnodes(AllData, "transport/plantdata/plant", NodeList)
forall(l in NodeList) do               ! Read plant data
  PLANTCAP(getstrattr(AllData,l,"name")):=
    getrealvalue(AllData, getnode(AllData,l,"capacity"))
  PLANTCOST(getstrattr(AllData,l,"name")):=
    getrealvalue(AllData, getnode(AllData,l,"cost"))
end-do
                                       ! Read routes data
getnodes(AllData, "transport/routes/route", NodeList)
forall(l in NodeList) do
  DISTANCE(getstrattr(AllData,l,"from"),getstrattr(AllData,l,"to")):=
    getrealattr(AllData,l,"distance")
  TRANSCAP(getstrattr(AllData,l,"from"),getstrattr(AllData,l,"to")):=
    getrealattr(AllData,l,"capacity")
end-do
                                       ! Read 'fuelcost' attribute
FUELCOST:= getrealattr(AllData, getnode(AllData, "transport"), "fuelcost")
```

In the model extract above we have used several simple XML path specifications to retrieve lists of nodes from the XML document. Such queries can take more complicated forms, including tests on node values (all plants with capacity >3500) or attributes (all routes to 'Scotland')—see the chapter on *mmxml* in the Mosel Language Reference Manual for further detail.

```
getnodes(AllData, "transport/plantdata/plant/capacity[number()>3500]/..", NodeList)
getnodes(AllData, "transport/routes/route[@to='Scotland']", NodeList)
```

We now also want to output the optimization results in XML format. As a first step, we create a root element 'solution' in the XML document `ResData`. In a well-formed XML document, all elements need to form a tree under the root element. All following nodes (containing the solution information per plant) are therefore created as element nodes under the 'solution' node. The objective function solution value and the execution date of the model are saved as attributes to the 'solution' tag. And finally, we use `save` to write an XML file or display a node with the (sub)tree formed by its children.

```
declarations
 ResData: xmldoc                                ! XML document
 Sol,Plant,Reg,Total: integer                   ! XML nodes
end-declarations

Sol:=addnode(ResData, 0, XML_ELT, "solution")     ! Create root node "solution"
setattr(ResData, Sol, "Objective", MinCost.sol)   ! Obj. value as attribute
setattr(ResData, Sol, "RunDate", text(datetime(SYS_NOW)))

forall(p in PLANT) do
  Plant:=addnode(ResData, Sol, XML_ELT, "plant")  ! Add a node to "solution"
  setattr(ResData, Plant, "name", p)              ! ... with attribute "name"
  forall(r in REGION | flow(p,r).sol>0) do
    Reg:=addnode(ResData, Plant, XML_ELT, "region") ! Add a node to "plant"
    setattr(ResData, Reg, "name", r)              ! ... with attribute "name"
    setvalue(ResData, Reg, flow(p,r).sol)         ! ... and solution value
  end-do
  Total:=addnode(ResData, Plant, "total",
                 sum(r in REGION)flow(p,r).sol)   ! Add node with total flow
end-do

save(ResData, "transportres.xml")   ! Save solution to XML format file
save(ResData, Sol, "")              ! Display XML format solution on screen
```

The Mosel code printed above will create a file `transportres.xml` with the following contents:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<solution Objective="8.1018e+07" RunDate="2012-10-17T14:00:50,664">
  <plant name="Corby">
    <region name="North">80</region>
    <region name="SEast">920</region>
    <region name="Midlands">2000</region>
    <total>3000</total>
  </plant>
  <plant name="Deeside">
    <region name="North">1450</region>
    <region name="SWest">1000</region>
    <region name="Midlands">250</region>
    <total>2700</total>
  </plant>
  <plant name="Glasgow">
    <region name="Scotland">2840</region>
    <region name="North">1270</region>
    <total>4110</total>
  </plant>
  <plant name="Oxford">
    <region name="SWest">1600</region>
    <region name="SEast">1900</region>
    <region name="Midlands">500</region>
    <total>4000</total>
  </plant>
</solution>
```

17.3.2.3 Generating HTML

An HTML file is generated and written out just like XML documents. The name of the root element in this case usually is 'html'. Below follows an extract of the code (example file `transport_html.mos`) that generates the HTML page shown in Figure 17.2. Notice the use of `copynode` to insert the same node/subtree at different positions in the XML document. By default, new nodes created with `addnode` are appended to the end of the node list of the specified parent node. This corresponds to using the value `XML_LASTCHILD` for the (optional) positioning argument of subroutines creating new nodes.

```
  declarations
   ResultHTML: xmldoc
   Root, Head, Body, Style, Title, Table, Row, Cell, EmptyCell: integer
  end-declarations

  Root:= addnode(ResultHTML, 0, XML_ELT, "html")       ! Root node
  Head:= addnode(ResultHTML, Root, XML_ELT, "head")    ! Element node
  Style:= addnode(ResultHTML, Head, XML_ELT, "style",
    "body {font-family: Verdana, Geneva, Helvetica, Arial, sans-serif;" +
    " color: 003f5f; background-color: d8e3e9 }\n" +
    "table td {background-color: e9e3db; color: 003f5f; text-align: right }\n" +
    "table th {background-color: f7c526; color: 003f5f}")
  setattr(ResultHTML, Style, "type", "text/css")       ! Set an attribute

  Body:= addnode(ResultHTML, Root, XML_ELT, "body")    ! Body of HTML page
  Title:= addnode(ResultHTML, Body XML_ELT, "h2", "Transportation Plan")

  Table:= addnode(ResultHTML, Body, XML_ELT, "table")  ! 'table' element
  setattr(ResultHTML, Table, "width", '100%')          ! Set some attributes
  setattr(ResultHTML, Table, "border", 0)
  Row:= addnode(ResultHTML, Table, XML_ELT, "tr")      ! Table row element
  EmptyCell:= addnode(ResultHTML, Row, XML_ELT, "td")  ! Table cell element
  setattr(ResultHTML, EmptyCell, "width", '5%')        ! Set an attribute
  Cell:= addnode(ResultHTML, Row, "td", "Total cost: " +
                 textfmt(MinCost.sol,6,2))  ! Table cell element with contents
  Cell:= addnode(ResultHTML, Cell, XML_DATA, "&#163;") ! DATA node
  Cell:= addnode(ResultHTML, Row, "td", text(datetime(SYS_NOW)))
  EmptyCell:=                                ! Node created by copying a node
    copynode(ResultHTML, EmptyCell, ResultHTML, Row, XML_LASTCHILD)
...
```

```
  save(ResultHTML, "transportres.html")      ! Write the HTML file
  save(ResultHTML, Table, "")                ! Display table def. on screen
```

The resulting HTML page might now look as shown in Figure 17.2.

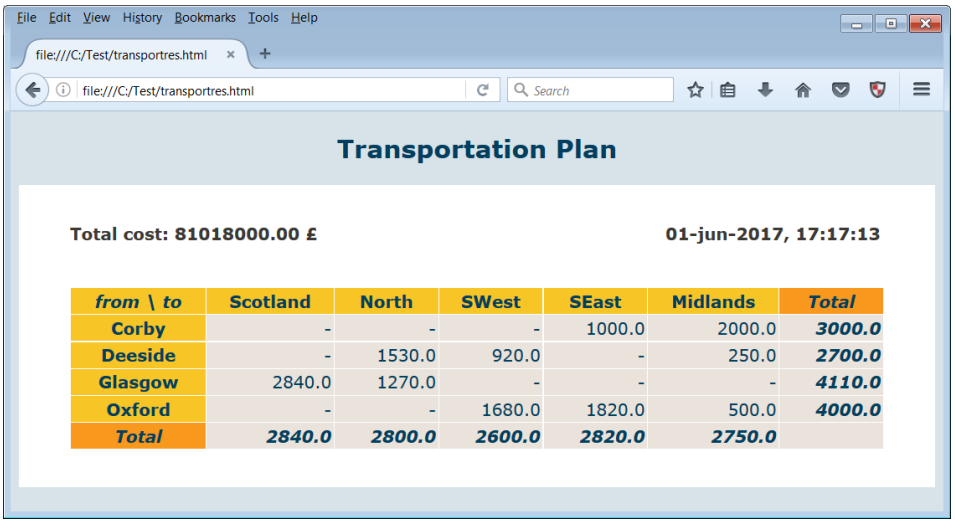


Figure 17.2: HTML page generated by Mosel

17.3.3 Xpress Insight

For embedding a Mosel model into Xpress Insight, we make a few edits to the model. All functionality that is needed to establish the connection between Mosel and Xpress Insight is provided by *mminsight* that now needs to be loaded. Since Xpress Insight manages the data scenarios, we only need to read in data from the original sources when loading the so-called *baseline scenario* into Xpress Insight (triggered by the test of `insightgetmode=INSIGHT_MODE_LOAD` in the model below), for model runs started from Xpress Insight (that is, in the case of `insightgetmode=INSIGHT_MODE_RUN`) the scenario data will be input directly from Xpress Insight at the insertion point marked with `insightpopulate`. For standalone execution (`insightgetmode=INSIGHT_MODE_NONE`) the model defaults to its original behavior, that is, reading the data from file followed by definition and solving of the optimization problem. Furthermore, the solver call to start the optimization is replaced by `insightminimize` / `insightmaximize`. Please also note that any model entities to be managed by Xpress Insight need to be declared as `public`—in the following code example this marker has been applied to the entire `declarations` block, alternatively it can be added to individual entity declarations.

```
model "Transport (Xpress Insight)"
 uses "mmxprs", "mminsight"

 public declarations
  REGION: set of string                ! Set of customer regions
  PLANT: set of string                 ! Set of plants

  DEMAND: array(REGION) of real        ! Demand at regions
  PLANTCAP: array(PLANT) of real       ! Production capacity at plants
  PLANTCOST: array(PLANT) of real      ! Unit production cost at plants
  TRANSCAP: dynamic array(PLANT,REGION) of real
                                       ! Capacity on each route plant->region
  DISTANCE: dynamic array(PLANT,REGION) of real
                                       ! Distance of each route plant->region
  FUELCOST: real                       ! Fuel cost per unit distance
```

```
 MaxCap: array(PLANT) of linctr        ! Capacity constraints
 flow: dynamic array(PLANT,REGION) of mpvar    ! Flow on each route
end-declarations

procedure readdata                     ! Data for baseline
 initializations from 'transprt.dat'
  DEMAND
  [PLANTCAP,PLANTCOST] as 'PLANTDATA'
  [DISTANCE,TRANSCAP] as 'ROUTES'
  FUELCOST
 end-initializations
end-procedure

case insightgetmode of
 INSIGHT_MODE_LOAD: do
    readdata                           ! Input baseline data and
    exit(0)                            ! stop the model run here
   end-do
 INSIGHT_MODE_RUN: insightpopulate     ! Inject scenario data and continue
 INSIGHT_MODE_NONE: readdata           ! Input baseline data and continue
 else
   writeln("Unknown run mode")
   exit(1)
end-case

! Create the flow variables that exist
forall(p in PLANT, r in REGION | exists(TRANSCAP(p,r)) ) create(flow(p,r))

! Objective: minimize total cost
MinCost:= sum(p in PLANT, r in REGION | exists(flow(p,r)))
           (FUELCOST * DISTANCE(p,r) + PLANTCOST(p)) * flow(p,r)

! Limits on plant capacity
forall(p in PLANT) MaxCap(p):= sum(r in REGION) flow(p,r) <= PLANTCAP(p)

! Satisfy all demands
forall(r in REGION) sum(p in PLANT) flow(p,r) = DEMAND(r)

! Bounds on flows
forall(p in PLANT, r in REGION | exists(flow(p,r)))
 flow(p,r) <= TRANSCAP(p,r)

insightminimize(MinCost)               ! Solve the problem through Xpress Insight
```

The handling of model entities by Xpress Insight can be configured via *annotations* (see Chapter 18 for detail), for example to define aliases to be displayed in the UI in place of the model entity names, or to select which data entities are to be treated as inputs or results respectively. The annotations defined by Xpress Insight form the category `insight`, the following model extract shows some example definitions for the 'Transport' problem, please refer to the *Xpress Insight Mosel Interface Manual* for a complete documentation.

```
!@insight.manage=input
public declarations
 !@insight.alias Customer regions
 REGION: set of string                 ! Set of customer regions
 !@insight.alias Plants
 PLANT: set of string                  ! Set of plants
 !@insight.alias Demand
 DEMAND: array(REGION) of real         ! Demand at regions
 !@insight.alias Production capacity
 PLANTCAP: array(PLANT) of real        ! Production capacity at plants
 !@insight.alias Unit production cost
 PLANTCOST: array(PLANT) of real       ! Unit production cost at plants
 !@insight.alias Capacity on each route
 TRANSCAP: dynamic array(PLANT,REGION) of real
 !@insight.alias Distance per route
 DISTANCE: dynamic array(PLANT,REGION) of real
```

```
 !@insight.alias Fuel cost per unit distance
 FUELCOST: real                          ! Fuel cost per unit distance
end-declarations

!@insight.manage=result
public declarations
 !@insight.alias Production capacity limits
 MaxCap: array(PLANT) of linctr          ! Capacity constraints
 !@insight.alias Amount shipped
 flow: dynamic array(PLANT,REGION) of mpvar    ! Flow on each route
 !@insight.hidden=true
 MincostSol: real
 !@insight.alias Total
 pltotal: array(PLANT) of real           ! Solution: production per plant
end-declarations
```

Xpress Insight expects models to be provided in compiled form, that is, as BIM files. An Xpress Insight *app archive* is a ZIP archive that contains the BIM file and the optional subdirectories `model_resources` (data files), `client_resources` (custom view definitions), and `source` (Mosel model source files). When developing an Insight app with Xpress Workbench, select the button to create the app archive or to publish the app directly to Insight.

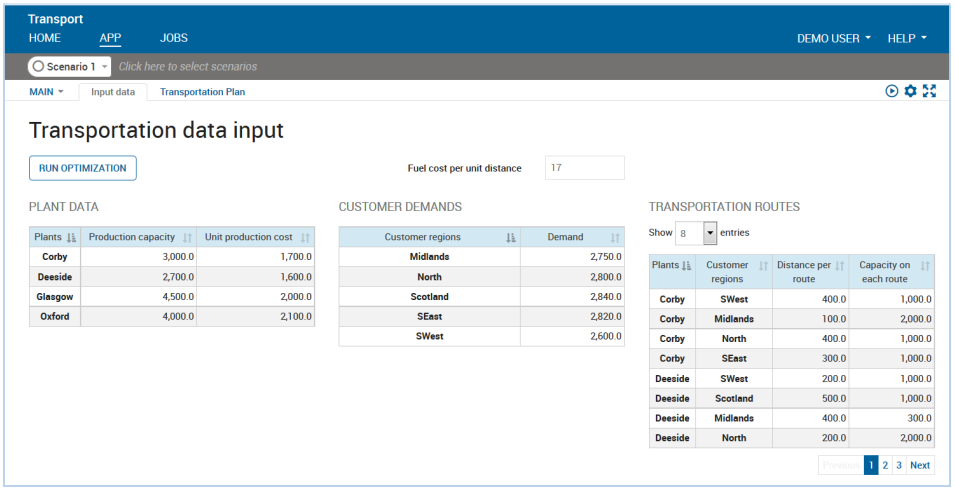


Figure 17.3: Xpress Insight web view showing a VDL view for the transportation problem

If we wish to deploy an optimization app to the Xpress Insight Web Client we need to take into account that besides the 'Entity browser' view that lists all managed model entities there are no default views: any visualization of input or result data needs to be explicitly implemented as *views*. The screenshot in Figure 17.3 shows a VDL (*View Definition Language*) view for the transportation example with editable input data and a 'Run' button that triggers re-solving of the optimization problem with the scenario data displayed on screen. Such views can be created via a drag-and-drop editor in Xpress Workbench (the screenshot in Figure 17.4 shows the design view of the VDL file that defines the webview in the previous figure). The complete set of files is provided in the app archive `transport_insight.zip`. Please refer to the Xpress Insight Developer Manual for further detail on view definition using the VDL markup language or the Xpress Insight Javascript API.

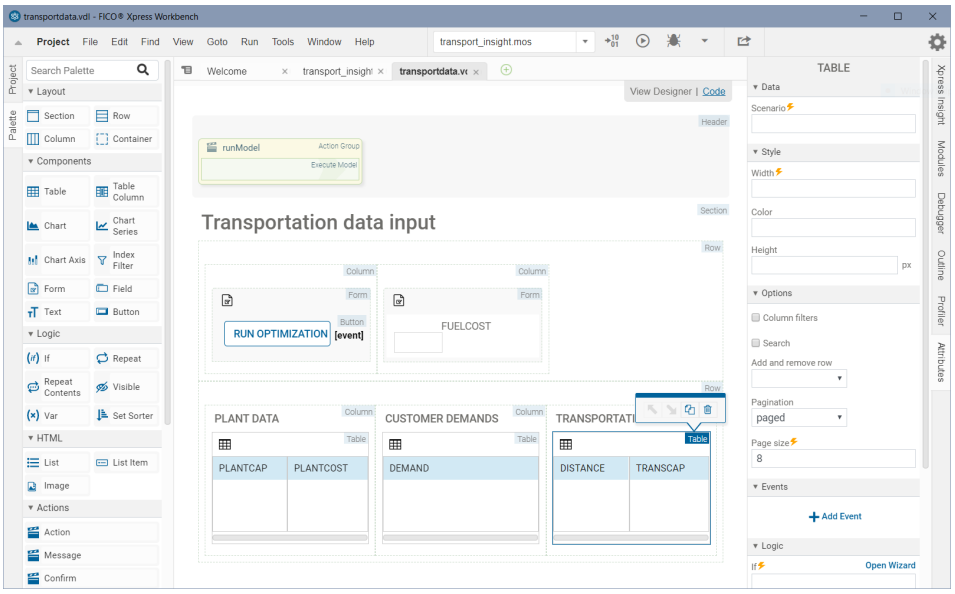


Figure 17.4: VDL view designer in Xpress Workbench

17.4 Solvers

In this user guide we have explained the basics of working with Mosel, focussing on the formulation of Linear and Mixed Integer Programming problems and related solution techniques. However, the Mosel language is not limited to certain types of Mathematical Programming problems.

The module *mmnl* extends the Mosel language with functionality for handling general non-linear constraints. This module (documented in the Mosel Language Reference Manual) does not contain any solver on its own. In combination with *mmxprs* you can use it to formulate and solve QCQP (Quadratically Constrained Quadratic Programming) problems through the QCQP solvers within Xpress Optimizer.

A special case of non-linear constraints are the so-called *general constraints*, that is, specific constraint relations that are recognized by MIP solvers but they require the module *mmxnlp* for their formulation in Mosel—see the discussion in Section 4.1.

The Mosel module *mmrobust* (documented in the whitepaper Robust Optimization with Xpress) extends the Mosel language with functionality for stating robust optimization problems and solving them via the Xpress Solvers.

Other solvers of FICO Xpress Optimization (Xpress NonLinear and Xpress Global for solving non-linear problems, and Xpress Kalis for Constraint Programming, CP) are provided with separate manuals and their own sets of examples. Please see the Xpress website for an overview of the available products.

With Mosel it is possible to combine several solvers to formulate hybrid solution approaches for solving difficult application problems. The whitepaper *Hybrid MIP/CP solving with Xpress Optimizer and Xpress Kalis*, available for download from the Xpress website, gives several examples of hybrid solving with LP/MIP and Constraint Programming.

Below follow a few examples for some of the solvers mentioned above.

17.4.1 QCQP solving with Xpress Optimizer

In Section 12.1 we have solved a quadratically constrained problem by a recursion algorithm. One might

be tempted to try solving this problem with a QCQP solver. Unfortunately, it does not meet the properties required by the definition of QCQP problems: problems solvable by QCQP must not contain equality constraints with quadratic terms (reformulation as two inequalities will not work either) since such constraints do not satisfy the convexity condition. We shall see below how to solve this problem with the general non-linear solver Xpress NonLinear (using SLP).

Let us therefore take a look at a different problem: the task is to distribute a set of points represented by tuples of x-/y-coordinates on a plane minimizing the total squared distance between all pairs of points. For each point i we are given a target location (CX_i, CY_i) and the (square of the) maximum allowable distance to this location, the (squared) radius R_i around this location.

In mathematical terms, we have two decision variables x_i and y_i for the coordinates of every point i. The objective to minimize the total squared distance between all points is expressed by the following sum.

$$\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \left((x_i - x_j)^2 + (y_i - y_j)^2 \right)$$

For every point i we have the following quadratic inequality.

$$(x_i - CX_i)^2 + (y_i - CY_i)^2 \leqslant R_i$$

The resulting Mosel model (file `airport_qp.mos`) looks thus.

```
model "airport"
 uses "mmxprs", "mmnl"

 declarations
  RN: range                      ! Set of airports
  R: array(RN) of real           ! Square of max. distance to given location
  CX,CY: array(RN) of real       ! Target location for each point
  x,y: array(RN) of mpvar        ! x-/y- coordinates
  LimDist: array(RN) of nlctr
 end-declarations

 initialisations from "airport.dat"
  CY  CX  R
 end-initialisations

! Set bounds on variables
 forall(i in RN) do
  -10<=x(i); x(i)<=10
  -10<=y(i); y(i)<=10
 end-do

! Objective: minimise the total squared distance between all points
 TotDist:= sum(i,j in RN | i<j) ((x(i)-x(j))^2+(y(i)-y(j))^2)

! Constraints: all points within given distance of their target location
 forall(i in RN)
  LimDist(i):= (x(i)-CX(i))^2+(y(i)-CY(i))^2 <= R(i)

 setparam("XPRS_verbose", true);
 minimise(TotDist);

 writeln("Solution: ", getobjval);
 forall(i in RN) writeln(i, ": ", getsol(x(i)), ", ", getsol(y(i)))

end-model
```

A QCQP matrix can be exported to a text file (option `""` for MPS or `"l"` LP format) through the `writeprob` function of Xpress Optimizer. That is, you need to add the following lines to your model after the problem definition:

```
setparam("XPRS_loadnames", true)
loadprob(TotDist)
writeprob("airport.mat","l")
```

A graphical representation of the result with *mmsvg*, obtained with the following lines of Mosel code, is shown in Figure 17.5.

```
! Set the size of the displayed graph
svgsetgraphviewbox(-10,-10,10,10)
svgsetgraphscale(20)

! Draw the target locations
svgaddgroup("T", "Target area", SVG_SILVER)
svgsetstyle(SVG_FILL,SVG_CURRENT)
forall(i in RN) svgaddcircle(CX(i), CY(i), sqrt(R(i)))

! Draw the solution points
svgaddgroup("S", "Solution", SVG_BLUE)
forall(i in RN) svgaddpoint(x(i).sol, y(i).sol)

! Output to file
svgsave("airport.svg")

! Update the display
svgrefresh
svgwaitclose
```

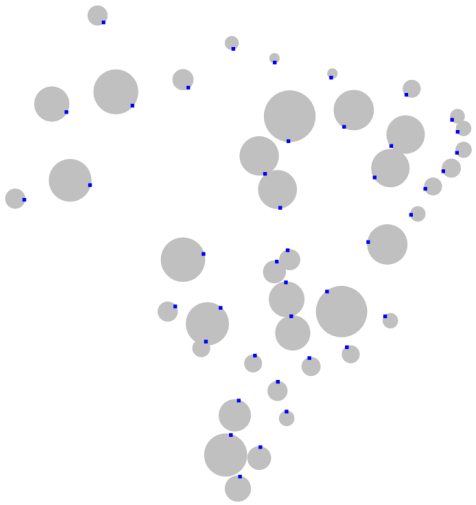


Figure 17.5: Result graphic in SVG format

17.4.2 Xpress NonLinear and Xpress Global

The following example solves the non-linear financial planning problem from Section 12.1 with Xpress NonLinear (using the SLP solver). The definition of constraints is straightforward (nonlinear constraints have the type `nlctr` with *mmxnlp*). Other functionality contributed by the module *mmxnlp* that appears in this model are the `setinitval` subroutine—used for setting start values for the decision variables— and the overloaded `minimize` subroutine for loading and solving nonlinear problems. Here, we simply need to become feasible and therefore use 0 as argument to `minimize`. Xpress used through the module *mmxnlp* automatically selects a solver among the installed solvers of the Xpress suite (Simplex, Barrier, Global, SLP, or Knitro) depending on the detected problem type. We know that

this problem can be solved by recursion and therefore preselect the SLP solver by setting the parameters XPRS_NLPSOLVER and XNLP_SOLVER; without these settings the solver selection will default to Xpress Global if it is available.

```
model "Recursion (NLP)"
 uses "mmxnlp"                        ! Use Xpress NonLinear

 declarations
  NT=6                                ! Time horizon
  QUARTERS=1..NT                      ! Range of time periods
  M,P,V: array(QUARTERS) of real      ! Payments

  interest: array(QUARTERS) of mpvar ! Interest
  net: array(QUARTERS) of mpvar       ! Net
  balance: array(QUARTERS) of mpvar   ! Balance
  rate: mpvar                         ! Interest rate
 end-declarations

 M:: [-1000, 0, 0, 0, 0, 0]
 P:: [206.6, 206.6, 206.6, 206.6, 206.6, 0]
 V:: [-2.95, 0, 0, 0, 0, 0]

 setinitval(rate, 0)                  ! Set initial values for variables
 forall(t in QUARTERS) setinitval(balance(t), 1)

                                      ! net = payments - interest
 forall(t in QUARTERS) net(t) = (M(t)+P(t)+V(t)) - interest(t)

                                      ! Money balance across periods
 forall(t in QUARTERS) balance(t) = if(t>1, balance(t-1), 0) - net(t)

                                      ! Interest rate
 forall(t in 2..NT) -(365/92)*interest(t) + balance(t-1) * rate = 0

 interest(1) = 0                      ! Initial interest is zero
 forall(t in QUARTERS) net(t) is_free
 forall(t in 1..NT-1) balance(t) is_free
 balance(NT) = 0                      ! Final balance is zero

 ! setparam("XNLP_VERBOSE",true)      ! Uncomment to see detailed output
 setparam("XPRS_NLPSOLVER", 1)        ! Use a local NLP solver
 setparam("XNLP_SOLVER", 0)           ! Select the SLP solver

 minimize(0)                          ! Solve the problem (get feasible)

                                      ! Print the solution
 writeln("\nThe interest rate is ", getsol(rate))
 write(strfmt("t",5), strfmt(" ",4))
 forall(t in QUARTERS) write(strfmt(t,5), strfmt(" ",3))
 write("\nBalances ")
 forall(t in QUARTERS) write(strfmt(getsol(balance(t)),8,2))
 write("\nInterest ")
 forall(t in QUARTERS) write(strfmt(getsol(interest(t)),8,2))
 writeln

end-model
```

The results displayed by this model are exactly the same as those we have obtained from the recursion algorithm in Section 12.1.

17.4.3 Xpress Kalis

Constraint Programming is an entirely different method of representing and solving problems compared to the Mathematical Programming approaches we have employed so far in this manual. Consequently, the Mosel module *kalis* defines its own set of types that behave differently from their

Mathematical Programming counterparts, including `cpvar`, `cpfloatvar` and `cpctr` for decision variables and constraints, but also `cpbranching` for search strategies (a standard constituent of CP models) and aggregate modeling objects such as `cptask` and `cpresource`.

Below we show the CP implementation of a binpacking example from the book 'Applications of optimization with Xpress-MP' (Section 9.4 'Backing up files'). The problem is to save sixteen files of different sizes onto empty disks of the same fixed capacity minimizing the total number of disks that are used.

Our model formulation remains close to the Mixed Integer Programming formulation and yet shows some specifities of Constraint Programming. Two sets of decision variables are used, $save_f$ indicating the choice of disk for file f and use_{fd} the amount of diskspace used by a file on a particular disk. Whereas the variables $save_f$ simply take integer values in a specified interval, each of the use_{fd} variables may take only two values, 0 or $SIZE_f$. For every file / disk combination we establish the logical relation

$$save_f = d \Leftrightarrow use_{fd} = SIZE_f$$

a constraint that cannot be stated in this form in a Mathematical Programming model. A second, so-called global constraint relation is used to state the `maximum` relation for calculating the number of disks used:

$$diskuse = maximum(save_{f \in DISKS})$$

And finally, the constraint limiting the capacity available per disk is a linear inequality that could occur in the same form in a Mathematical Programming model.

$$\forall d \in DISKS : \sum_{f \in FILES} use_{fd} \leqslant CAP$$

This is the complete Mosel CP model for the binpacking problem.

```
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
 end-declarations

 initializations from '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)

 setparam("kalis_default_lb", 0)

 declarations
  save: array(FILES) of cpvar       ! Disk a file is saved on
  use: array(FILES,DISKS) of cpvar  ! Space used by file on disk
  diskuse: cpvar                    ! Number of disks used
 end-declarations

! Set variable domains
 forall(f in FILES) setdomain(save(f), DISKS)
```

```
 forall(f in FILES, d in DISKS) setdomain(use(f,d), {0, SIZE(f)})

! Correspondence between disk choice and space used
 forall(f in FILES, d in DISKS) equiv(save(f)=d, use(f,d)=SIZE(f))

! Limit the number of disks used
 diskuse = maximum(save)

! Capacity limit of disks
 forall(d in DISKS) sum(f in FILES) use(f,d) <= CAP

! 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: ", getsol(sum(f in FILES) use(f,d)))
 end-do

end-model
```

This implementation is one of several possible formulations of this problem as a CP model. An alternative and generally more efficient model being the formulation as a cumulative scheduling problem, where the disks are represented by a single resource of discrete capacity, the files to save correspond to tasks of duration 1 with a resource requirement defined by the file size. The objective in this case is to minimize the duration of the schedule (= number of disks used). The interested reader is refered to the Xpress Kalis Mosel User Guide for a detailed discussion of this problem.

17.5 Date and time data types

The module *mmsystem* of the standard distribution of Mosel defines the types `date` (calendar day: day, month, and year), `time` (time of the day in milliseconds), and `datetime` (combination of the first two) for working with date and time related data in Mosel models. We show here some examples of

- reading and writing dates and times from/to file,
- formatting dates and times,
- using sets of constant dates and times for indexing arrays,
- transformation from/to the underlying numerical representation,
- applying operations (comparison, addition, difference, sorting),
- enumerating dates and times.

17.5.1 Initializing dates and times

The following line prints out the current date and time (using the default format):

```
 writeln("Today: ", date(SYS_NOW), ", current local time: ", time(SYS_NOW),
         "UTC time: ", gettime(datetime(timestamp)) )
```

When we wish to read data from a file, the formatting of dates and times needs to be adapted to the format used in the file. For example, consider the following data file (`datetime.dat`)

```
Time1:  "4pm"
Time2:  "16h00"
Time3:  "16:00:00"

Date1:  "2-20-2002"
Date2:  "20/02/02"
Date3:  "20-Feb-2002"
```

A Mosel model reading in this data file may look thus (file `dates.mos`).

```
declarations
 t: time
 d: date
end-declarations

setparam("timefmt", "%h%p")             ! h: hours in 1-12, p: am/pm
setparam("datefmt", "%m-%d-%y")         ! m: month, d: day, y: year

initializations from "datetime.dat"
 t as "Time1"
 d as "Date1"
end-initializations

writeln(d, ", ", t)

setparam("timefmt", "%Hh%0M")           ! H: hours in 0-23, M: minutes
setparam("datefmt", "%0d/%0m/%0Y")      ! Y: year in 0-99
                                        ! 0: fill spaces with '0'

initializations from "datetime.dat"
 t as "Time2"
 d as "Date2"
end-initializations

writeln(d, ", ", t)

setparam("timefmt", "%H:%0M:%0S")       ! S: seconds
setparam("datefmt", "%d-%N-%y")         ! N: use month names

initializations from "datetime.dat"
 t as "Time3"
 d as "Date3"
end-initializations

writeln(d, ", ", t)
```

For the encoding of date and time format strings please refer to the documentation of the parameters `datefmt` and `timefmt` in the 'Mosel Language Reference Manual'.

`Date3` in this example uses a month *name*, not a number. The default 3-letter abbreviations of month names can be changed (*e.g.*, translated) by redefining the parameter `monthnames`. For instance, a date written in French, such as

```
Date4:  "20 fevrier 2002"
```

is read by the following Mosel code:

```
setparam("datefmt", "%d %N %y")

setparam("monthnames", "janvier fevrier mars avril mai juin juillet " +
         "aout septembre octobre novembre decembre")

initializations from "datetime.dat"
 d as "Date4"
end-initializations
```

```
writeln(d)
```

In the examples of this section we have used Mosel's standard text format for reading and writing dates and times. These data types can also be used when accessing spreadsheets or databases through Mosel's ODBC connection or the software-specific interfaces for Oracle and MS Excel. The whitepaper *Using ODBC and other database interfaces with Mosel* documents some examples of accessing date and time data in spreadsheets and databases.

Note: When initializing or constructing dates Mosel does not control whether they correspond to an actual calendar day (*e.g.*, 0-valued or negative day and month counters are accepted). The validity of a date or time can be tested with the function `isvalid`. For example, the following code extract

```
d:= date(2000,0,0)
writeln(d, " is a valid date: ", if(isvalid(d), "true", "false"))
```

results in this output:

```
2000-00-00 is a valid date: false
```

17.5.2 Dates and times as constants

It is possible to use the types 'date', 'time', 'datetime' as index sets for arrays if the elements of the set are flagged as being `constant`. The effect of such a declaration as constant is illustrated by the following code snippet (taken from the example file `dates.mos`). Entities such as `someday` in the example below that receive a value directly in the declarations block via '=' also are constants, their value can not be changed or re-assigned.

```
declarations
  someday=date(2020,3,24)           ! A constant date
  SD: set of date                   ! Set of date references
  SDC: set of constant date         ! Set of constant date references
  AD: dynamic array(SDC) of real    ! Array indexed by 'date' type
end-declarations

! Operations on a set of dates
SD:= {date(2020,3,24), date(2020,3,24)+1}
writeln("Is someday in SD? ", someday in SD)             ! Output: false
writeln("Next day in SD? ", someday+1 in SD)             ! Output: false
SD+= {date(2020,3,24), date(2020,3,24)+1}
writeln("SD after addition: ", SD, " size=", SD.size)    ! Output: size=4

! Operations on a set of constant dates
SDC:= {date(2020,3,24),date(2020,3,24)+1}
writeln("Is someday in SDC? ", someday in SDC)           ! Output: true
writeln("Next day in SDC? ", someday+1 in SDC)           ! Output: true
SDC+= {date(2020,3,24), date(2020,3,24)+1}
writeln("SDC after addition: ", SDC, " size=", SDC.size) ! Output: size=2
```

The example shown here only mentions the type 'date', but the `constant` declaration is also applicable to the types 'time' and 'datetime'.

17.5.3 Conversion to and from numbers

In some cases it might be necessary to use the numerical representation in the place of a date or time. In the following Mosel extract we wish to define an array `YEARS` that is indexed by a set of dates. In this example we show how to use as index values the numerical representation that is obtained by applying `getasnumber` to the dates (this function returns an integer, the Julian Day number = number of days elapsed since 1/1/1970; if the argument is a time `getasnumber` returns the number of milliseconds since midnight). By applying `date` to the numerical representation it is converted back to the original date.

With this Mosel code

```
declarations
 Dates: set of date
 YEAR: array(NDates: set of integer) of integer
end-declarations

setparam("datefmt", "")                 ! Use the default format

initializations from "datetime.dat"
 Dates
end-initializations

writeln("Dates: ", Dates)
forall(dd in Dates) YEAR(getasnumber(dd)):= getyear(dd)
writeln("YEAR: ", YEAR)
forall(n in NDates) writeln(date(n))    ! Mapping back to original dates
```

and the following data

```
Dates: [ "1999-1-21" "2000-2-22" "2002-3-23" "2005-4-24" "2010-5-25"]
```

we obtain this output:

```
Dates: {1999-01-21,2000-02-22,2002-03-23,2005-04-24,2010-05-25}
YEAR: [(10612,1999),(11009,2000),(11769,2002),(12897,2005),(14754,2010)]
1999-01-21
2000-02-22
2002-03-23
2005-04-24
2010-05-25
```

Similarly to what is shown here, function `getasnumber` can be used with 'time' and 'datetime', the backwards conversion being carried out by `time` or `datetime` respectively.

17.5.4 Operations and access functions

The following Mosel model extract (`dates.mos`) shows some operations on dates and times, including sorting lists of dates or times, difference between two dates or times, and addition of constants to obtain an enumeration of dates or times.

```
declarations
 t: time
 d: date
 now1,now2: datetime
 DNAMES: array(1..7) of string
 TList: list of time
 DList: list of date
end-declarations

! Difference between dates
writeln("February 2004 had ", date(2004,3,1)-date(2004,2,1), " days.")

! Retrieve the weekday
DNAMES:: (1..7)["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                "Saturday", "Sunday"]
writeln("1st January 2000 was a ", DNAMES(getweekday(date(2000,1,1))))

! Difference between times
now1:= datetime(SYS_NOW)
wait(1)                                 ! Delay model execution for 1 second
now2:= datetime(SYS_NOW)
writeln("Elapsed time: ", now2-now1, "ms")
```

```
! Enumeration / addition to 'time'
 setparam("timefmt", "%.h.%0M%p")
 t:= time(11,0)
 forall(i in 1..5) do
  writeln(t)
  t+=30*60*1000                          ! Add 30 minutes
 end-do

! Enumeration / addition to 'date'
 setparam("datefmt", "%.d/%0m/%0Y")
 d:= date(2005,12,20)
 forall(i in 1..5) do
  writeln(d)
  d+=14                                  ! Add 14 days
 end-do

! Sorting lists of dates and times
 setparam("datefmt", "")                 ! Revert to default date format
 DList:= [date(2021,1,1),date(1900,1,2),date(2020,3,24)]
 writeln("Orig.  DL=", DList)
 qsort(SYS_UP, DList)
 writeln("Sorted DL=", DList)

 setparam("timefmt", "")                 ! Revert to default time format
 TList:= [time(12,0),time(10,30),time(16,15),time(8,45)]
 writeln("Orig.  TL=", TList)
 qsort(SYS_UP, TList)
 writeln("Sorted TL=", TList)
```

Executing this model produces the following output.

```
February 2004 had 29 days.
1st January 2000 was a Saturday
Elapsed time: 1.006ms
11.00am
11.30am
12.00pm
12.30pm
 1.00pm
20/12/05
 3/01/06
17/01/06
31/01/06
14/02/06
Orig.  DL=[2021-1-1,1900-01-02,2020-03-24]
Sorted DL=[1900-01-02,2020-03-24,2021-1-1]
Orig.  TL=[12:00:00,10:30:00,16:15:00,8:45:00]
Sorted TL=[8:45:00,10:30:00,12:00:00,16:15:00]
```

17.6 Text handling and regular expressions

The module *mmsystem* provides a large set of text handling functionality, including

- the types `text`, `parsectx`, and `textarea`
- text formatting routines (number format, upper/lower case)
- parsing routines
- regular expressions

In the following subsections we show some examples of text handling with Mosel, for a full description of the available functionality please refer to the chapter *mmsystem* of the 'Mosel Language Reference Manual'.

17.6.1 `text` vs. `string`

Although apparently denoting similar concepts, the purpose and usage recommendations for the types `string` and `text` in Mosel models are quite distinct: any `string` defined in a model is added to the model's names dictionary and is only freed at termination of the model run, this is not the case for model objects of the type `text`. The type `string` therefore should be used whenever it is a question of identifying objects, so in particular for index sets.

The type `text` is in general the more appropriate choice for descriptive or editable texts, including reporting or logging messages, and any texts generated via (partial) copies or concatenation. A `text` object can be altered, allowing for a considerably wider set of operations (such as insertion, deletion) in comparison with strings. Furthermore, with the I/O driver `text:` a public text object can be used as input or output file in a model (see Section 17.1.2).

It is, however, not always possible to draw a clear line between where to use `string` or `text`. A number of module subroutines therefore define multiple versions, accepting both, string or text arguments. Note further that if required, Mosel automatically converts from the type `string` to `text`, but not the other way round.

17.6.2 Parsing text

In the example below we configure the global parser settings to read real numbers from a text that has fields separated by commas.

```
declarations
  values: list of real
  comma=getchar(",",1)              ! ASCII value for ","
end-declarations

txt:= text(", , 123.4 , 345.6 ,")

! Parsing without context
setparam("sys_sepchar", comma)      ! Comma as separation character
setparam("sys_trim", true)          ! Trim blanks around separation character
while (nextfield(txt)) do           ! Get next field
  values+= [parsereal(txt)]         ! Read a real number from the field
  writeln("Read up to position ", getparam("sys_endparse"))
end-do
writeln("Values read: ", values)    ! Output: [0,0,123.4,345.6,0]
```

The same behavior can be achieved with a parser context—here we do not modify any global settings, which has the advantage of preventing possible interactions with other parser settings that may be used elsewhere in our model.

```
declarations
  pctx: parsectx
  values: list of real
  comma=getchar(",",1)              ! ASCII value for ","
end-declarations

txt:= text(", , 123.4 , 345.6 ,")

! Parsing real numbers with context
setsepchar(pctx, comma)             ! Comma as separation character
settrim(pctx, true)                 ! Trim blanks around separation character
while (nextfield(txt,pctx)) do      ! Get next field
  values+= [parsereal(txt, pctx)]   ! Read a real number from the field
  writeln("Read up to position ", pctx.endparse)
end-do
writeln("Values read: ", values)    ! Output: [0,0,123.4,345.6,0]
```

When implementing data handling for optimization applications, it is good practice to add error handling to the parsing loop, for example to check whether the fields are formatted as expected:

```
pctx.endparse:=0                    ! Start at the beginning of text
pctx.sepchar:=comma                 ! Comma as separation character
pctx.trim:=true                     ! Trim blanks around separation character
while (nextfield(txt,pctx)) do      ! Get next field
  if getchar(txt, pctx.endparse)=comma or pctx.endparse>=txt.size then
    values+=[0.0]                   ! The field is empty
  else
    r:=parsereal(txt, pctx)         ! Read a real number from the field
    if getsysstat=0 then values+= [r]
    else
      writeln("Malformed field contents at position ", pctx.endparse,
      " (", copytext(txt, pctx.endparse,pctx.endparse+2), ")")
    end-if
  end-if
  writeln("Read up to position ", pctx.endparse)
end-do
writeln("Values read: ", values)    ! Output: [0,0,123.4,345.6,0]
```

One might also choose to work with multiple parser contexts (*e.g.* using an 'inner' context `pctxi` for reading some part of each field from the original text—here an integer number that is read from a string containing a real).

```
declarations
  pctx,pctx1: parsectx
  ivalues: list of integer
  comma=getchar(",",1)              ! ASCII value for ","
end-declarations

txt:= text(", , 123.4 , 345.6 ,")

setsepchar(pctx, comma)             ! Comma as separation character
settrim(pctx, true)                 ! Trim blanks around separation character
while (nextfield(txt,pctx)) do      ! Get next field
  tt:=parsetext(txt, pctx)          ! Get contents of the field
  pctxi.endparse:=1                 ! Reset start to beginning of the text
  i:=parseint(tt,pctxi)             ! Read an integer number from the field
  if getsysstat=0: ivalues+= [i]
  writeln("Read up to position ", pctx.endparse)
end-do
writeln("Values read: ", ivalues)    ! Output: [123,345]
```

17.6.3 Regular expressions

A *regular expression* (in the following abbreviated to *regex*) is a sequence of characters that form a search pattern. Regex are used to describe or match a set of strings according to certain syntax rules. Mosel supports the Basic Regular Expressions syntax (BRE) and the Extended Regular Expressions syntax (ERE) of the POSIX standard, the implementation of regular expression matching relies on the TRE library.

Here are some examples of regular expression matching and replacement with some explanations of the meaning of the employed regex—for a complete description of the supported regex syntax the reader is refered to the documentation of the TRE library (see http://laurikari.net/tre), another useful resource are the examples provided on the page en.wikipedia.org/wiki/Regular_expression.

The following example (`regex.mos`) displays all strings containing 'My' that occur in a text. The first matching statement uses BRE syntax, it displays all strings starting with 'My' irrespective of upper/lower case spelling (option `REG_ICASE`). The second matching statement uses ERE syntax (option `REG_EXTENDED`) to retrieve all strings containing 'My' other than at their beginning. We have chosen to retrieve different individual portions of the matching string (specified via the parantheses in the regular

expression statement) the positions of which are stored in their order of occurrence into the array `m` (of type `textarea`)

```
declarations
  m: array(range) of textarea
  t: text
end-declarations
t:="MyValue=10,Sometext Mytext MoretextMytext2, MYVAL=1.5 mYtext3"
m(0).succ:=1
while (regmatch(t, '\<My\(\w*\)', m(0).succ, REG_ICASE, m))
  writeln("Word starting with 'My': ", copytext(t,m(0)))
! Output:  MyValue  Mytext  MYVAL  mYtext3

m(0).succ:=1
while (regmatch(t, '\w+((My)(\w*))', m(0).succ, REG_ICASE+REG_EXTENDED, m))
  writeln("String containing 'My' (not at beginning): ",
   copytext(t,m(0)), " (", copytext(t,m(1)), "=", copytext(t,m(2)) ,
   "+", copytext(t,m(3)), ")")
! Output:  MoretextMytext2 (Mytext2=My+text2)
```

The special characters used in the formulation of the regular expressions above have the following meaning: `\<` marks the beginning of a word, `\w` denotes alphanumeric or underscore characters, `*` means 0 or more times and + stands for 1 or more times.

The following Mosel code snippet shows how to replace matching expressions in a text that contains dates with different formats:

```
t:="date1=20/11/2010,date2=1-Oct-2013,date3=2014-6-30"
numr:= regreplace(t, '([[:digit:]]{4})-([01]?[[:digit:]])-([0-3]?[[:digit:]])',
              '\3/\2/\1', 1, REG_EXTENDED)
if numr>0 then
 writeln(numr, " replacements: ", t)
end-if
```

This is the output produced by the code above:

```
1 replacements: date1=20/11/2010,date2=1-Oct-2013,date3=30/6/2014
```

There are alternative ways of stating the same regular expression with BRE or ERE syntax, for example:

```
numr:= regreplace(t, '\(\d\{4\}\)-\([01]\{0,1\}\d\)-\([0-3]\{0,1\}\d\)',
         '\3/\2/\1' )
numr:= regreplace(t, '(\d{4})-([01]{0,1}\d)-([0-3]{0,1}\d)',
         '\3/\2/\1', 1, REG_EXTENDED )
```

In these replacement statements we have used the following special characters for stating regular expressions: `\d` or `[:digit:]` indicates a numerical character, square brackets contain a set of possible character matches, `{M,N}` means minimum M and maximum N match count and `?` stands for 0 times or once.

CHAPTER 18

Annotations

Annotations are *meta data* expressed in a Mosel source file (model or package) that are stored in the resulting BIM file after compilation. This additional information is either global or associated with public globally declared objects (including subroutines). Annotations do not have any direct impact on the model itself as they are treated like comments. Typical uses of annotations include model documentation or application configuration information.

Mosel annotations have the following format:

- a single-line annotation starts with '!@' and a name

```
!@doc.name This is my document title
```

- multi-line annotations are surrounded by '(!@' and '!)'

```
(!@mynote Some annotation text.
Another line of text.
!)
```

- '!@' is followed by the *annotation name* (identifier)
  - no space between '!' and '@' characters
  - space is allowed between '@' and the name
- value assignment operators are ' ' (space), ':', or '='
  - no space between annotation name and operator

```
!@mynote contents for 'mynote'
!@ another=contents of 'another'
!@third: contents of 'third'
```

- association with symbols:

```
!@mynote This annotation is applied to all objects declared below
public declarations
  val: integer            !@doc.descr Explanation of 'val'
  !@doc.descr Text associated with 'msg'
  msg: string             ! A standard comment
  !@mynote2 This annotation is ignored (no associated object)
end-declarations
```

Annotations are organized in *categories*. A category groups a set of annotations and other categories (or sub-categories). For example

```
doc.name
```

will be used to select the annotation `name` member of the `doc` category. Predefined category names include `mc` (Mosel compiler) and `doc` (model documentation). Models can also define/employ new annotations and categories—these must be valid Mosel identifiers, that is, their names can only use alpha-numeric symbols and `'_'`.

We show some examples of the `doc` category in Section 18.2 below. The category `mc` is used to pass information to the compiler during the compilation, including the (optional) declaration of new annotations with the `mc.def` annotation or the definition of aliases, such as

```
!@mc.def descr alias doc.descr insight.descr
```

that redirects onto two different annotations (see section 'Annotations' of the Mosel Language Reference Manual for further detail).

18.1 Accessing annotations

Annotations can be retrieved from the model itself during its execution or before/after execution from the calling program (using the Mosel libraries). The following example `annottest.mos` shows how to retrieve an annotation that is defined in the same model. The subroutine `getannotations` is defined by *mmreflect*, an overloaded version (defined by *mmjobs*) takes an additional first argument of type `Model` indicating the submodel it is to be applied to instead of the model itself.

```
model "Using annotations"
 uses "mmreflect"

 public declarations
(!@.
  @value.first 5
  @value.second 0
  @descr A scalar value
 !)
  myint: integer
  AnnNames: set of string          !@descr Set of annotation names
  Ann: array(string) of string     !@descr Annotation values
  mytxt: text                      !@descr Default input data file
 end-declarations

 !@descr Annotations test
 !@furtherinfo Simply displays all defined global or specific annotations

 ! Get all global annotations defined in this model:
 getannotations("", "", AnnNames, Ann)
 writeln("Global annotations:")
 forall(a in AnnNames) writeln("  ", a, " = ", Ann(a))

 ! Get all annotations for "myint":
 getannotations("myint", "", AnnNames, Ann)
 writeln("Annotations defined for 'myint':")
 forall(a in AnnNames) writeln("  ", a, " = ", Ann(a))

 ! Retrieve all annotations starting with 'value.' and that are
 ! associated to 'myint'
 getannotations("myint", "value.", AnnNames, Ann)
 writeln("'value' annotations for 'myint':")
 forall(a in AnnNames) writeln("  ", a, " = ", Ann(a))
end-model
```

Running this model produces output like the following:

```
Global annotations:
  .descr = Annotations test
  .furtherinfo = Simply displays all defined global or specific annotations
Annotations defined for 'myint':
  .descr = A scalar value
  value.first = 5
  value.second = 0
'value' annotations for 'myint':
  value.first = 5
  value.second = 0
```

And here is an example how to retrieve annotations into a C program (file `annotdisplay.c`):

```
#define MAXANN 100

int main()
{
 XPRMmodel mod;
 void *ref;
 const char *symb;                    /* A model object name */
 const char *ann[MAXANN*2];           /* List of annotations */
 int i,n;

 if(XPRMinit()) return 1;             /* Initialize Mosel */
 if((mod=XPRMloadmod("annottest.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

/* Retrieve and display global annotations */
 n=XPRMgetannotations(mod,NULL,NULL,ann,MAXANN*2);
 printf("Global annotations (total: %d):\n", n/2);
 for(i=0;i<n && i<MAXANN;i+=2)
  printf("   %s:%s\n",ann[i],(ann[i+1]!=NULL)?ann[i+1]:"");

/* Retrieve and display all annotations associated with model objects */
 printf("Annotations associated with objects:\n");
 ref=NULL;
 while((symb=XPRMgetnextanident(mod,&ref))!=NULL)
 {
  n=XPRMgetannotations(mod,symb,NULL,ann,MAXANN*2);
  printf(" %s->\n",symb);
  for(i=0;i<n && i<MAXANN;i+=2)
   printf("   %s:%s\n",ann[i],(ann[i+1]!=NULL)?ann[i+1]:"");
 }

/* Retrieve and display annotations for model object 'myint' */
 n=XPRMgetannotations(mod,"myint",NULL,ann,MAXANN*2);
 printf("Annotations defined for 'myint' (total: %d):\n", n/2);
 for(i=0;i<n && i<MAXANN;i+=2)
  printf("   %s:%s\n",ann[i],(ann[i+1]!=NULL)?ann[i+1]:"");

 return 0;
}
```

The corresponding Java code looks as follows:

```
public class annotdisplay
{
 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  XPRMAnnotation ann[];                        // List of annotations

  mosel = new XPRM();                          // Initialize Mosel
  mod = mosel.loadModel("annottest.bim");      // Load a BIM file

// Retrieve and display global annotations
```

```
  ann=mod.getAnnotations("");
  System.out.println("Global annotations (total: "+ ann.length +"):");
  for(int i=0;i<ann.length;i++) System.out.println("   "+ann[i]);

// Retrieve and display all annotations associated with model objects
  System.out.println("Annotations associated with objects:");
  for(XPRMIdentifiers ids=mod.annotatedIdentifiers(); ids.hasNext();)
  {
   XPRMIdentifier id=(XPRMIdentifier)ids.next();
   ann=mod.getAnnotations(id,"");
   System.out.println(" "+id.getName()+"->");
   for(int i=0;i<ann.length;i++) System.out.println("   "+ann[i]);
  }

// Retrieve and display annotations for model object 'myint'
  ann=mod.getAnnotations("myint","");
  System.out.println("Annotations defined for 'myint' (total: "+ ann.length +"):");
  for(int i=0;i<ann.length;i++) System.out.println("   "+ann[i]);
 }
}
```

Notice that the host application only needs to load the BIM file (and not necessarily run a model) in order to be able to retrieve the annotations.

18.2 *moseldoc*: Generating model documentation

The Mosel compiler reserves a special treatment to annotations belonging to the `doc` category. This annotation category will only be included into the BIM file if the source (model or package) file is compiled with the option `–D`, such as

```
mosel comp –D mymodel.mos
```

To enable the -D compiler option in Xpress Workbench open the *Run* menu and select the entry *Compiler Options*, enable the *Generate Doc Annotations* option, and confirm with *Save*. When you next click the *Compile* button  in the Workbench toolbar, the resulting BIM file will contain documentation annotations.

The tool *moseldoc* can be applied to the resulting BIM file to generate an XML file that is then processed into a set of HTML pages:

```
moseldoc mymodel                    ! Generates HTML and XML
moseldoc –html mymodel              ! HTML output only
moseldoc –o mydir –html mymodel     ! Specify HTML output directory
moseldoc –xml mymodel               ! XML output only
moseldoc –ixml mymodel              ! XML file for inclusion (omitting header+root)
moseldoc –f mymodel                 ! Force output overwrite
```

Here are some examples of how to use the documentation annotations (example file `annotdoc.mos`):

- Document structure:

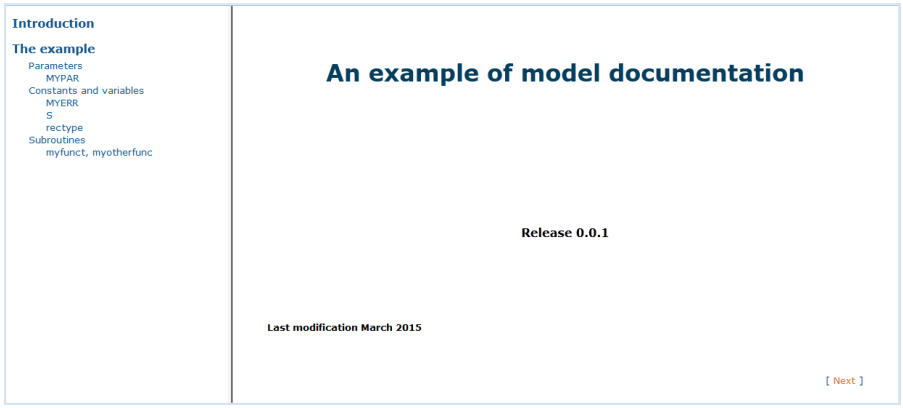
```
    (!@doc.
     @title An example of model documentation
     @version 0.0.1
     @date March 2015

     @chapter Introduction
     @p
     This model needs to be compiled with the <tt>-D</tt> compiler option
     to include the documentation annotations into the BIM.
     The <tt>moseldoc</tt> program takes the resulting BIM file as input.
```

```
!)

!@doc.chapter The example
!@doc.section Parameters
```

The resulting cover page and table of contents generated by *moseldoc* look as follows (the contents listing also refers to the entity and subroutine annotations shown in the following items):



- Documenting parameters:

```
parameters
 !@doc.descr Short parameter description
 !@doc.value v1 possible value
 !@doc.value v2 another possible value
 !@doc.info Some more explanation (longer text)
 MYPAR="some text"
end-parameters
```

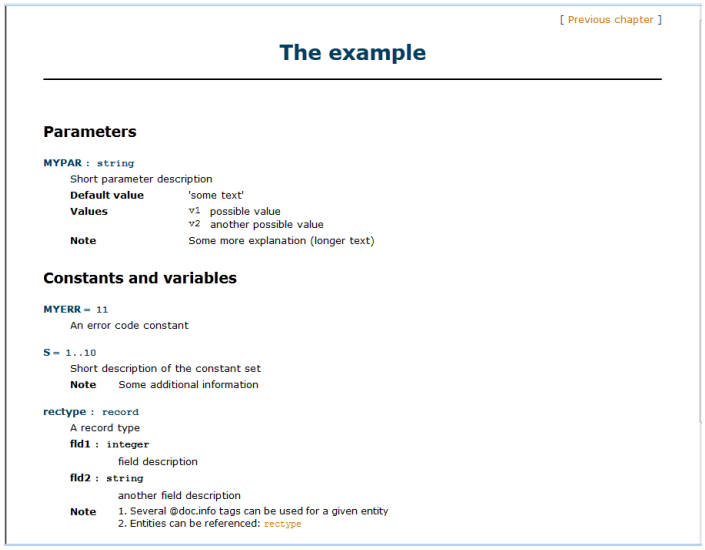
- Documenting entity declarations (type definitions, constants, variables):

```
!@doc.section Constants and variables
public declarations
 (!@doc.
   @descr Short description of the constant set
   @info Some additional information
 !)
 S=1..10

 !@doc.descr An error code constant
 MYERR=11

 (!@doc.
   @descr A record type
   @recflddescr fld1 field description
   @recflddescr fld2 another field description
   @info Several @doc.info tags can be used for a given entity
   @info Entities can be referenced: <entRef>rectype</entRef>
 !)
 rectype=public record
  fld1:integer
  fld2:string
 end-record
end-declarations
```

This is the HTML page generated by *moseldoc* from these parameter and entity annotations:


[Previous chapter]

The example

Parameters

MYPAR : string
Short parameter description
Default value 'some text'
Values v1 possible value
v2 another possible value
Note Some more explanation (longer text)

Constants and variables

MYERR = 11
An error code constant

S = 1..10
Short description of the constant set
Note Some additional information

rectype : record
A record type
fld1 : integer
field description
fld2 : string
another field description
Note 1. Several @doc.info tags can be used for a given entity
2. Entities can be referenced: rectype

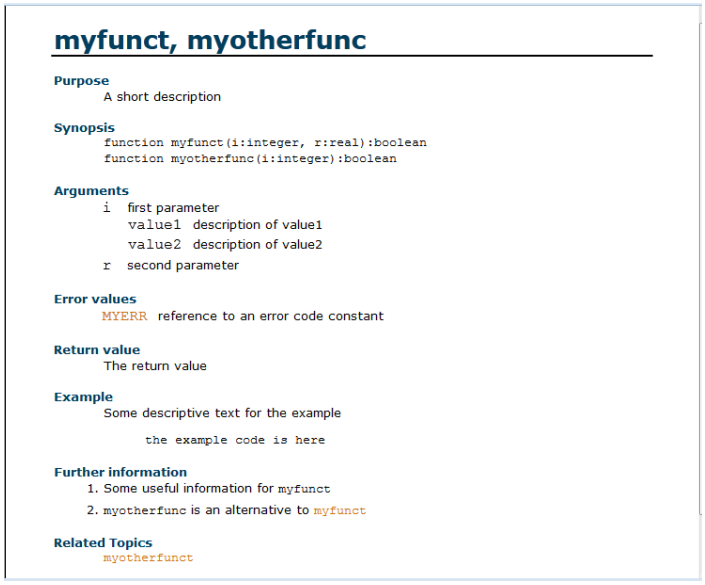
- Documenting subroutines:

```
!@doc.section Subroutines

(!@doc.
  @descr A short description
  @param i first parameter
  @paramval i value1 description of value1
  @paramval i value2 description of value2
  @param r second parameter
  @err MYERR reference to an error code constant
  @return The return value
  @example Some descriptive text for the example
  @example [SRC]
  the example code is here
  @info Some useful information for <tt>myfunct</tt>
  @related <fctRef>myotherfunct</fctRef>
!)
public function myfunct(i:integer,r:real):boolean
 returned:=i>r
end-function

!@doc.group myfunct
!@doc.info <tt>myotherfunc</tt> is an alternative to <fctRef>myfunct</fctRef>
public function myotherfunc(i:integer):boolean
 returned:=true
end-function
```

The HTML subroutine documentation page generated by *moseldoc* from these annotations is shown here:



myfunct, myotherfunc

Purpose
A short description

Synopsis
```
function myfunct(i:integer, r:real):boolean
function myotherfunc(i:integer):boolean
```

Arguments
`i` first parameter
`value1` description of value1
`value2` description of value2
`r` second parameter

Error values
`MYERR` reference to an error code constant

Return value
The return value

Example
Some descriptive text for the example

```
the example code is here
```

Further information
1. Some useful information for `myfunct`
2. `myotherfunc` is an alternative to `myfunct`

Related Topics
`myotherfunct`

- Relocating documentation contents:

```
!@doc.section A section title
!@doc.relocate newlocref
 ...   ! All doc annotations defined here will be inserted at marker 'newlocref'
!@doc.relocate
 ...   ! All subsequently defined doc annotations remain where they are

!@doc.section Destination location
!@doc.location newlocref
```

- Excluding contents from the generated documentation:

```
!@doc.autogen=false
public declarations
  ...  ! No automatic generation of annotations: any non-annotated entities
       ! defined here will not be included in the documentation
end-declarations
!@doc.autogen=true
  ...  ! All subsequently defined entities will get processed

! The following procedure will not be included into the documentation
!@doc.descr Ignored by the documentation
!@doc.ignore
public procedure hiddenproc
 a:=1
end-procedure

!@doc.ignore
public declarations
  ...  ! All entities defined here will not be included in the documentation
end-declarations
```

The *moseldoc* tool is provided in compiled form with FICO Xpress Optimization release distributions, its source and a complete example of its use (file `mddoc.mos`) that can be found in the subdirectory `examples/mosel/Moseldoc` of Xpress installations or online in the Xpress Examples Repository.

V. Remote invocation of Mosel

Overview

All previous sections of this manual assume that you are working with a standard installation of Xpress on your local computer. However, a local installation of Xpress is not a requirement when working with Mosel. The examples in this part show how to use the *Mosel remote invocation library XPRD* for building applications requiring the Xpress technology that run from environments where Xpress is not installed—including architectures for which Xpress is not available.

XPRD is a self-contained library (*i.e.* with no dependency on the usual Xpress libraries) that relies on the *Mosel Distributed Framework* (module *mmjobs*, see Section 17.2). The examples in this part are introductory examples of some of the most commun programming tasks when working with a remote installation of Xpress, namely

- starting Mosel instances (locally or on remote hosts)
- compiling, loading, running, and interrupting Mosel models remotely
- redirection of standard streams
- sending and receiving events
- retrieving data from a Mosel model

Further examples, particularly of more advanced uses, are discussed in the whitepaper *Multiple models and parallel solving with Mosel* and also in the *Advanced Evaluators' Guide*. Both documents are provided with their examples as a part of the Xpress distribution. For a complete documentation of the XPRD library the reader is referred to the *XPRD Reference Manual*.

The first chapter (Chapter 19) of this part introduces the C version of XPRD. The Java versions of the same examples are described in Chapter 20.

Chapter 19
XPRD C

The program example `runprimedistr.c` below shows how to run the model `prime.mos` remotely using the XPRD C library (NB: this program corresponds to the *mmjobs* distributed computing example `runprimedistr.mos` from Section 17.2.4). At first sight, the reader might be reminded of the Mosel C libraries presented in Chapter 13. However, there are two major additions besides the change of the prefixes from `XPRM` to `XPRD`:

- We need to *connect* to a remote machine and create a new Mosel instance (`XPRDmosel`) prior to working with any Mosel models. Remote machines are specified by their name or IP address, the empty string in the present example indicates that we want to use the local machine.
- The submodel is executed in an independent process and we therefore need to *wait* for its termination.

In summary, the standard execution sequence for Mosel models of *compile load run* is augmented to *connect – compile load run wait* in the context of distributed computing (this remark equally applies to submodels launched via *mmjobs*).

Instead of simply waiting for the submodel to terminate, the program below waits for 2 seconds and if no termination event message has been received from the Mosel model, it is stopped by the application. After termination of the submodel (either by finishing its calculations within less than 2 seconds or stopped by the parent model) the application reports the full event information and also displays the termination status and the exit value of the Mosel model. Unloading a model explicitly as shown here is only really necessary in larger applications that continue after the termination of the submodel, so as to free the memory used by it.

```
#include <stdio.h>
#include <stdlib.h>
#include "xprd.h"

int main(int argv,char *args[])
{
  XPRDcontext xprd;
  XPRDmosel moselInst;
  XPRDmodel modPrime, evsender;
  double evvalue;
  int evclass;

  xprd=XPRDinit();              /* Create an XPRD context */

  /* Open connection to a remote node: "" means the node running this program */
  moselInst=XPRDconnect(xprd, "", NULL, NULL, NULL, 0);
                                /* Compile the model file */
  XPRDcompmod(moselInst, "", "rmt:prime.mos", "rmt:prime.bim", "");
                                /* Load BIM into the remote instance */
  modPrime=XPRDloadmod(moselInst, "rmt:prime.bim");

  XPRDrunmod(modPrime, "LIMIT=50000");  /* Start execution and */
```

```
  XPRDwaitevent(xprd,2);        /* wait 2 seconds for an event */

  if (XPRDqueueempty(xprd)==1)  /* No event has been sent... */
  {
   printf("Model too slow: stopping it!\n");
   XPRDstoprunmod(modPrime);    /* ... stop the model, then wait */
   XPRDwaitevent(xprd,-1);
  }

  XPRDgetevent(xprd, &evsender, &evclass, &evvalue);    /* Get the event */
  printf("Event value: %g sent by model %d\n", evvalue, XPRDgetnum(evsender));
  printf("Exit status: %d\n", XPRDgetstatus(modPrime));
  printf("Exit code  : %d\n", XPRDgetexitcode(modPrime));

  XPRDunloadmod(modPrime);      /* Unload the model */
  XPRDdisconnect(moselInst);    /* Disconnect remote instance */
  XPRDfinish(xprd);             /* Terminate XPRD */

  remove("prime.bim");          /* Clean up temporary files */

  return 0;
}
```

In this example, we assume that the model source is saved on the local machine running XPRD, and the BIM file is written back to this machine (indicated by the I/O driver prefix `rmt:` in the 'compile' and 'load' functions that are executed on the remote instance). Alternatively, we might choose to save the BIM file on the remote machine, *e.g.* in memory (`shmem:primebim`) or in Mosel's temporary directory (`tmp:prime.bim`).

19.1 Exchanging data with the model

A typical programming task when working with remote models is the retrieval of results into the calling application. In this section we show how to use XPRD functionality for retrieving data that is written by a Mosel model in an `initializations` block into the XPRD program. The choice of the method for exchanging data usually depends on the particular system setup (write access rights) and the volume of data to be communicated (memory usage). Data in Mosel format can be output

1. on the remote machine running Mosel
   (a) in memory
   (b) as a physical file
2. on the local machine running XPRD
   (a) in memory
   (b) as a physical file

Case 1a is implemented in the program version `runprimeiodistr.c` printed below. Case 1b is obtained by removing the `shmem:` prefix from the file name, for example, the setting

```
OUTPUTFILE=bin:tmp:resdata
```

will create a file `resdata` in Mosel's temporary directory. Cases 2a and 2b could use the setting

```
OUTPUTFILE=bin:rmt:resdata
```

For the implementation of case 2b we simply replace the calls to the XPRD remote file access functions by the standard C library functions `fopen`, `fread`, and `fclose`. Somewhat more work is required for

the implementation of case 2a: the program needs to define an XPRD file manager to handle the data in memory—an example implementation is provided in the file `runprimeiodistr2.c`.

All implementation versions share the use of the `bin:` I/O driver and they all define the same function `show_solution` that decodes Mosel's binary format and displays the solution values. Using Mosel's binary format is recommended (though not a necessity) in distributed applications—it is platform-independent and uses less space than the standard text format.

The program example of the previous section stops the Mosel model with a call to `XPRDstoprunmod`. We now replace this hard stop by sending the user event 'STOPMOD' to the submodel: instead of immediately terminating the submodel this event is intercepted by the submodel and makes it interrupt its calculations and write out the current solution. To make sure that the submodel is actually running at the point where we sent the 'STOPMOD' event, we have also introduced a 'MODREADY' event sent from the submodel to its parent to indicate the point of time when it starts the calculations (with heavy operating system loads the actual submodel start may be delayed).

The Mosel model `primeio.mos` remains the same as shown in Section 17.2.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xprd.h"
#include "bindrv.h"

#define STOPMOD 2            /* Identifier for "Stop submodel" user event */
#define MODREADY 3           /* Identifier for "Submodel ready" user event */

int main(int argv,char *args[])
{
  XPRDcontext xprd;
  XPRDmosel moselInst;
  XPRDmodel modPrime, evsender;
  double evvalue;
  int evclass;
  XPRDfile f;

  xprd=XPRDinit();              /* Create an XPRD context */
                                /* Open connection to a remote node:
              "" means the node running this program */
  moselInst=XPRDconnect(xprd, "", NULL, NULL, NULL, 0);
                                /* Compile the model file */
  XPRDcompmod(moselInst, "", "rmt:primeio.mos", "tmp:primeio.bim", "");
                                /* Load the bim file into the remote instance */
  modPrime=XPRDloadmod(moselInst, "tmp:primeio.bim");

                                /* Disable submodel output */
 XPRDsetdefstream(moselInst, modPrime, XPRD_F_WRITE, "null:");

                                /* Start execution */
  XPRDrunmod(modPrime, "LIMIT=50000,OUTPUTFILE=bin:shmem:resdata");
  XPRDwaitevent(xprd,0);        /* Wait for an event */
  XPRDgetevent(xprd, &evsender, &evclass, &evvalue);    /* Get the event */
  if (evclass != MODREADY)      /* Check the event class */
  {
    printf("Problem with submodel run");
    return 1;
  }

  XPRDwaitevent(xprd,2);        /* Wait 2 seconds for an event */

  if (XPRDqueueempty(xprd)==1)  /* No event has been sent */
  {
   printf("Model too slow: stopping it!\n");
   XPRDsendevent(modPrime, STOPMOD, 0);    /* Stop the model, then  */
   XPRDwaitevent(xprd,-1);      /* wait for its termination */
  }
```

```
  /* Open the output file, retrieve and display the solution data */
  f=XPRDfopen(moselInst, "shmem:resdata", XPRD_F_BINARY|XPRD_F_INPUT, NULL,0);
  show_solution(my_read,f);
  XPRDfclose(f);

  XPRDunloadmod(modPrime);       /* Unload the model */
  XPRDdisconnect(moselInst);     /* Disconnect remote instance */
  XPRDfinish(xprd);              /* Terminate XPRD */

  return 0;
}
```

Once the submodel has terminated we read its solution from the location specified in the model parameter `OUTPUTFILE` and display the results. The subroutine `show_solution` uses functions from the bindrv library that is provided with XPRD to decode Mosel's binary format. The output file read by this routine has the same structure as the correponding text file in Mosel format, for example:

```
NumP: 6
SPrime: [ 2 3 5 7 11 13 ]
```

Since XPRD and bindrv expect different signatures for their reading functions, we also define a wrapper function `my_read`.

```
/**** Wrapper function for 'bindrv' ****/
static size_t my_read(void *buf, size_t size, size_t nmemb, void *ctx)
{
 size_t s,a;
 long t;

 s=size*nmemb;
 a=0;
 while(s>0)
 {
  t=XPRDfread(ctx,(char*)buf+a,s);
  if(t<=0) break;
  else
  {
   a+=t;
   s-=t;
  }
 }
 return a/size;
}

/**** Using bindrv: Decode the binary file and display its contents ****/
void show_solution(size_t (*doread)(void *,size_t,size_t,void*), void *rctx)
{
 s_bindrvctx bdrv;
 int *solarr;
 int size,i,n;
 char *str;

 bdrv=bindrv_newreader(doread,rctx);  /* Initialize binreader */

 i=size=0;
 solarr=NULL;
 while(bindrv_nexttoken(bdrv)>=0)
 {
  bindrv_getctrl(bdrv,&n);         /* 'label'  (marker) */
  bindrv_getstring(bdrv,&str);        /* Read a string */
  if(strcmp(str,"NumP")==0)
  {
   free(str);
   bindrv_getint(bdrv,&size);         /* Read an integer */
   printf("( %d prime numbers)\n", size);
   if(size>0)                         /* Prepare array to receive values */
```

```
    solarr=malloc(sizeof(int)*size);
   else
    break;
  }
  else
  if(strcmp(str,"SPrime")==0)
  {
   free(str);
   bindrv_getctrl(bdrv,&n);       /* [  (start marker) */
   while(bindrv_nexttoken(bdrv)==BINDRV_TYP_INT)
   {                                  /* Read integers */
    bindrv_getint(bdrv,&(solarr[i++]));
   }
   bindrv_getctrl(bdrv,&n);        /* ]  (end marker) */
  }
  else
  {
   printf("Unexpected label: %s\n", str);
   free(str);
   exit(1);
  }
 }

 bindrv_delete(bdrv);              /* Release bin reader */

/* Print the set of prime numbers */
 printf("Prime numbers={");
 for(i=0;i<size;i++)
  printf(" %d",solarr[i]);
 printf("}\n");

 free(solarr);                     /* Clean up */
}
```

CHAPTER 20

XPRD Java

For the remote execution of Mosel models we need to augment the standard execution sequence for Mosel models (that we have seen, for example, in Section 14.1) of *compile load run* to the sequence *connect – compile load run wait* (this remark equally applies to submodels launched via *mmjobs*). The meaning of these additions is the following:

- We need to *connect* to a remote machine and create a new Mosel instance (`XPRDmosel`) prior to working with any Mosel models. Remote machines are specified by their name or IP address, the empty string in the example below indicates that we want to use the local machine.
- The submodel is executed in an independent process and we therefore need to *wait* for its termination.

The program example `runprimedistr.java` below shows how to run the model `prime.mos` using the XPRD Java library. If the submodel has not terminated after 2 seconds (*i.e.*, not termination message has been received from this model), then it is stopped by the application. After termination of the submodel (either by finishing its calculations within less than 2 seconds or stopped by the parent model) the application reports the full event information and also displays the termination status and the exit value of the Mosel model. Unloading a model explicitly as shown here is only really necessary in larger applications that continue after the termination of the submodel, so as to free the memory used by it.

```
import com.dashoptimization.*;
import java.lang.*;
import java.io.*;

public class runprimedistr
{
 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();
  XPRDMosel moselInst;
  XPRDModel modPrime;
  XPRDEvent event;

  moselInst=xprd.connect("");    // Open connection to remote nodes
                                 // "" means the node running this program

                                 // Compile the model file on remote instance
  moselInst.compile("", "rmt:prime.mos", "rmt:prime.bim");

                           // Load the bim file into remote instance
  modPrime=moselInst.loadModel("rmt:prime.bim");

  modPrime.execParams = "LIMIT=50000";
  modPrime.run();                // Start execution and
  xprd.waitForEvent(2);          // wait 2 seconds for an event

  if (xprd.isQueueEmpty())       // No event has been sent...
```

```
  {
   System.out.println("Model too slow: stopping it!");
   modPrime.stop();              // ... stop the model, then wait
   xprd.waitForEvent();
  }
                                 // An event is available: model finished
  event=xprd.getNextEvent();
  System.out.println("Event value: " + event.value +
                     " sent by model " + event.sender.getNumber());
  System.out.println("Exit status: " + modPrime.getExecStatus());
  System.out.println("Exit code  : " + modPrime.getResult());

  moselInst.unloadModel(modPrime);  // Unload the submodel
  moselInst.disconnect();           // Terminate the connection

  new File("prime.bim").delete();   // Clean up temporary files
 }
}
```

The model source file `prime.mos` used by this example is saved on the local machine running XPRD, and the BIM file is written back to this machine (indicated by the I/O driver prefix `rmt:` in the 'compile' and 'load' function calls that are executed on the remote instance). Alternatively, we might choose to save the BIM file on the remote machine, *e.g.* in memory (`shmem:primebim`) or in Mosel's temporary directory (`tmp:prime.bim`).

20.1 Exchanging data with the model

An application that processes a Mosel model typically needs to retrieve some (result) data from the model for reporting or further treatment. Besides exchanging data via external sources (*e.g.* databases), Mosel offers a number of possibilities for directly retrieving data into an XPRD program. The choice of the method for exchanging data usually depends on the particular system setup (write access rights) and the volume of data to be communicated (memory usage). Data in Mosel format written in an `initializations` block can be output

1. on the remote machine running Mosel
   (a) in memory
   (b) as a physical file
2. on the local machine running XPRD
   (a) in memory
   (b) as a physical file

Case 1a is implemented in the program version `runprimeiodistr.java` printed below. Case 1b is obtained by removing the `shmem:` prefix from the file name, for example, the setting

```
OUTPUTFILE=bin:tmp:resdata
```

will create a file `resdata` in Mosel's temporary directory. Cases 2a and 2b could use the setting

```
OUTPUTFILE=bin:rmt:resdata
```

For the implementation of case 2b we simply replace the XPRD remote file access by standard Java file access, for example

```
resdata=new FileInputStream("resdata");
```

Somewhat more work is required for the implementation of case 2a: the program needs to define an XPRD file manager to handle the data in memory—an example implementation is provided in the file `runprimeiodistr2.java`.

All implementation versions share the use of the `bin:` I/O driver and they all define the same function `showSolution` that decodes Mosel's binary format and displays the solution values. Using Mosel's binary format is recommended (though not a necessity) in distributed applications—it is platform-independent and uses less space than the standard text format.

The program version printed below introduces two user events to achieve more precise time measures for the remote process: the hard stop of the Mosel model is replaced by sending the user event 'STOPMOD' to the submodel: instead of immediately terminating the submodel this event is intercepted by the submodel and makes it interrupt its calculations and write out the current solution. To make sure that the submodel is actually running at the point where we sent the 'STOPMOD' event, we have also introduced a 'MODREADY' event sent from the submodel to its parent to indicate the point of time when it starts the calculations (with heavy operating system loads the actual submodel start may be delayed).

We work with the Mosel model `primeio.mos` from Section 17.2.3.

```
import com.dashoptimization.*;
import java.lang.*;
import java.util.*;
import java.io.*;

public class runprimeiodistr
{
 static final int STOPMOD = 2;   // Identifier for "Stop submodel" user event
 static final int MODREADY = 3;  // Identifier for "Submodel ready" user event

 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();          // Initialize XPRD
  XPRDMosel moselInst;
  XPRDModel modPrime;
  XPRDEvent event;
  InputStream resdata;

  moselInst=xprd.connect("");    // Open connection to remote nodes
                                 // "" means the node running this program

                                 // Compile the model file on remote instance
  moselInst.compile("", "rmt:primeio.mos", "tmp:primeio.bim");

                          // Load the bim file into remote instance
  modPrime=moselInst.loadModel("tmp:primeio.bim");

                                 // Disable submodel output
  modPrime.setDefaultStream(modPrime.F_OUTPUT, "null:");

  modPrime.execParams = "LIMIT=50000,OUTPUTFILE=bin:shmem:resdata";
  modPrime.run();                // Start execution and
  xprd.waitForEvent();           // ...wait for an event
  event=xprd.getNextEvent();     // Retrieve the event
  if (event.eventClass != MODREADY)   // Check the event class
  {
    System.out.println("Problem with submodel run");
    System.exit(1);
  }

  xprd.waitForEvent(2);          // Let the submodel run for 2 seconds

  if (xprd.isQueueEmpty())       // No event has been sent...
  {
   System.out.println("Model too slow: stopping it!");
```

```
   modPrime.sendEvent(STOPMOD, 0);    // ... stop the model, then
   xprd.waitForEvent();               // wait for its termination
  }

  // Open the output file, retrieve and display the solution data
  resdata=moselInst.openForReading("shmem:resdata", moselInst.F_BINARY);
  showSolution(resdata);
  resdata.close();

  moselInst.unloadModel(modPrime);    // Unload the submodel
  moselInst.disconnect();             // Terminate the connection
 }
}
```

After termination of the model run, the XPRD application reads the solution data from the location specified in the model parameter `OUTPUTFILE` and displays the results. Below follows the implementation of the function `showSolution` that uses an instance of `BinDrvReader` to decode Mosel's binary format (the bindrv library is provided with XPRD). A binary format files has the same structure as the correponding text file in Mosel format, for example:

```
NumP: 6
SPrime: [ 2 3 5 7 11 13 ]
```

Each data entry starts with a label (string followed by a colon), followed either by a single, scalar data value, or a list of data values surrounded by square brackets.

```
// **** Decode the binary stream and display its contents ****
 static void showSolution(InputStream inbuf) throws Exception
 {
  BinDrvReader bdrv=new BinDrvReader(inbuf);  // Initialize binreader
  String label;
  ArrayList<Integer> setP=new ArrayList<Integer>();

  while(bdrv.nextToken()>=0)
  {
   bdrv.getControl();          // 'label'  (marker)
   label=bdrv.getString();        // Read a string
   if(label.equals("NumP"))
   {                              // Read an integer
    System.out.println("(" + bdrv.getInt() + " prime numbers.)");
   }
   else
   if(label.equals("SPrime"))
   {
    bdrv.getControl();         // [  (start marker)
    while(bdrv.nextToken()==BinDrvReader.TYP_INT) // or ] at end of list
    {                             // Read integers
     setP.add(Integer.valueOf(bdrv.getInt()));
    }
    bdrv.getControl();         // ]  (end marker)
   }
   else
   {
    System.out.println("Unexpected label: "+label);
    System.exit(0);
   }
  }

 // Display the contents of the set 'SPrime'
  Iterator<Integer> iprime=setP.iterator();
  System.out.print("Prime numbers={");
  while(iprime.hasNext())
  {
   Integer p=iprime.next();
   System.out.print(" "+p);
  }
```

```
  System.out.println(" }");
 }
```

Appendix

APPENDIX A

Mosel Language overview

A.1 Structure of a Mosel model

A Mosel model (text file with extension `.mos`) has the form

```
model model_name

  Compiler directives

  Parameters

  Body

end-model
```

Compiler directives

- Options are specified as a *compiler directive*, at the beginning of the model
- Options include `explterm`, which means that each statement must end with a semi-colon, and `noimplicit`, which forces all objects to be declared

```
options explterm
options noimplicit
```

- `uses` statements are also compiler directives

```
uses "mmxprs", "mmodbc"
```

- Can define a version number for your model

```
version 1.0.0
```

- Another set of compiler directives serves for the definition and configuration of namespaces

```
namespace mynsp
nssearch myns2
```

Run-time parameters

- Scalars (of type `integer`, `real`, `boolean`, or `string`) with a specified default value
- Their value may be reset when executing the model
- Use `initializations from` for inputting structured data (arrays, sets,...)
- At most one `parameters` block per model

Model body

- Model statements other than compiler directives and parameters, including any number of

  - `declarations`
  - `initializations from`/`initializations to`
  - functions and procedures

Implicit declaration

- Mosel does *not* require all objects to be declared
- Simple objects can be used without declaring them, if their type is obvious
- Use the `noimplicit` option to force all objects to be declared before using them (see item *Compiler directives* above)

Mosel statements

- Can extend over several lines and use spaces
- However, a line break acts as an expression terminator
- To continue an expression, it must be cut after a symbol that implies continuation (*e.g.* `+ - , * (`)

A.2 Data structures

`array`, `set`, `list`, `record` and any combinations thereof, *e.g.*,

```
S: set of list of integer
A: array(range) of set of real
```

Arrays

Array: collection of labeled objects of a given type where the label of an array entry is defined by its index tuple

```
declarations
  A: array(1..5) of real
  B: array(range, set of string) of integer
  x: array(1..10) of mpvar
end-declarations

A:: [4.5, 2.3, 7, 1.5, 10]
A(2):= 1.2
B:: (2..4,["ABC", "DE", "KLM"])[15,100,90,60,40,15,10,1,30]
```

Sets

Set: collection of objects of the same type without establishing an order among them (as opposed to arrays and lists)
Set elements are unique: if the same element is added twice the set still only contains it once.

```
declarations
  S: set of string
  R: range
end-declarations

S:= "A", "B", "C", "D"
R:= 1..10
```

Lists

List: collection of objects of the same type
A list may contain the same element several times. The order of the list elements is specified by construction.

```
declarations
  L: list of integer
  M: array(range) of list of string
end-declarations

L:= [1,2,3,4,5]
M:: (2..4)[['A','B','C'], ['D','E'], ['F','G','H','I']]
```

Records

Record: finite collection of objects of any type
Each component of a record is called a *field* and is characterized by its name and its type.

```
declarations
  ARC: array(ARCSET:range) of record
    Source,Sink: string      ! Source and sink of arc
    Cost: real               ! Cost coefficient
  end-record
end-declarations

ARC(1).Source:= "B"
ARC(3).Cost:= 1.5
```

User types

User types are treated in the same way as the predefined types of the Mosel language. New types are defined in `declarations` blocks by specifying a type name, followed by =, and the definition of the type.

```
declarations
  myreal = real
  myarray = array(1..10) of myreal
  COST: myarray
end-declarations
```

Union types

Union: container capable of holding an object of one of a predefined set of types. Defined by specifying the set of compatible types or the predefined union type `any`.

```
declarations
  u: string or real              ! Scalar accepting 'string' or 'real'
  a: any                         ! Scalar accepting any type
 ! Defining a type name for the union of the 4 basic Mosel types:
  basictype = string or integer or real or boolean
  U: array(range) of basictype   ! Array of union type 'basictype'
end-declarations
```

A.3 Selection statements

if ... end-if

```
if c=1 then
  writeln('c equals 1')
end-if
```

if ... else ... end-if

```
if c=1 then
  writeln('c equals 1')
else
  writeln('c does not equal 1')
end-if
```

if ... elif ... else ... end-if

```
if c=1 then
  writeln('c equals 1')
elif c1 then
  writeln('c is bigger than 1')
else
  writeln('c is smaller than 1')
end-if
```

case ... end-case

```
case c of
  1,2 : writeln('c equals 1 or 2')
  3   : writeln('c equals 3')
  4..6: do
         writeln('c is in 4..6')
         writeln('c is not 1, 2 or 3')
        end-do
else
  writeln('c is not in 1..6')
end-case
```

A.4 Loops

forall

```
forall(f in FAC, t in TIME)
  make(f,t) = MAXCAP(f,t)

forall(t in TIME) do
  use(t) = MAXUSE(t)
  buy(t) = MAXBUY(t)
end-do
```

with

equivalent to a `forall` loop stopped after the first iteration

```
with f='F1', t=1 do
  make(f,t) = MAXCAP(f,t)
end-do
```

while

```
i := 1
while (i = 10) do
  write(' ', i)
  i += 1
end-do
```

repeat ... until

```
i := 1
repeat
  write(' ', i)
  i += 1
until (i  10)
```

break, next

- `break` jumps out of the current loop
- `break` *n* jumps out of *n* nested loops (where *n* is a positive integer)
- `next` jumps to the beginning of the next iteration of the current loop

counter

- Use the construct `as counter` to specify a counter variable in a bounded loop (*i.e.*, `forall` or aggregate operators such as `sum`). At each iteration, the counter is incremented

```
cnt:=0.0
writeln("Average of odd numbers in 1..10: ",
        (sum(cnt as counter, i in 1..10 | isodd(i)) i) / cnt)
```

A.5 Operators

Assignment operators

```
i := 10
i += 20        ! Same as i := i + 20
i -=  5        ! Same as i := i - 5
```

Assignment operators with linear constraints

```
C := 5*x + 2*y = 20
D := C + 7*y
```

then `D` is

```
D := 5*x + 9*y - 20
```

The constraint type is dropped with `:=`

```
C := 5*x + 2*y = 20
C += 7*y
```

then `C` is

```
C := 5*x + 9*y = 20
```

The constraint type is retained with +=, −=

Arithmetic operators

standard:	`+ - * /`
power:	`^`
int. division/remainder:	`mod div`
sum:	`sum(i in 1..10) ...`
product:	`prod(i in 1..10) ...`
minimum/maximum:	`min(i in 1..10) ...`
count:	`count(i in 1..10 | isodd(i))`

Linear and non-linear expressions

Decision variables can be combined into linear or non-linear expressions using the arithmetic operators

- module *mmxprs* only works with linear constraints, so no `prod`, `min`, `max`, ...
- other solver modules, *e.g.*, *mmnl*, *mmxnlp*, also accept (certain) non-linear expressions

Logical operators

constants:	`true, false`
standard:	`and, or, not`
AND:	`and(i in 1..10) ...`
OR:	`or(i in 1..10) ...`
comparison:	`<, >, =, <>, <=, >=`

Set operators

constants:	`{'A', 'B'}`
union:	`+`
union:	`union(i in 1..10) ...`
intersection:	`*`
intersection:	`inter(i in 1..10) ...`
difference:	`-`

Set comparison operators

subset:	`Set1 <= Set2`
superset:	`Set1 >= Set2`
equals:	`Set1 = Set2`
not equals:	`Set1 <>Set2`
element of:	`"Oil5" in Set1`
not element of:	`"Oil5" not in Set1`

List operators

constants:	`[1, 2, 3]`
concatenation:	`+, sum`
truncation:	`-`
equals:	`L1 = L2`
not equals:	`L1 <>L2`
enumeration:	`i in L` (within `forall`, `sum` *etc.*)

Union and reference operators

testing properties of unions:

```
! u declared as a union type, such as 'any'
writeln(u is array of integer)
writeln(u is not procedure)
```

'reference to' operator:

```
L:= [-cos,-sin,-arctan,-exp]
forall(f in L) writeln(f(0.5))
```

A.6 Built in functions and procedures

The following is a list of built in functions and procedures of the Mosel language (excluding modules). Functions return a value; procedures do not.

Dynamic array handling

```
create    exists    delcell   isdynamic
```

Freeze (finalize) a dynamic set

```
finalize
```

Rounding functions

```
ceil      floor     round     abs
```

Mathematical functions

```
exp       log       ln        sqrt
cos       sin       arctan
isodd
```

Special real values

```
isfinite  isinf     isnan
```

Random number generator

```
random    setrandseed
```

Minimum/maximum of a list of values

```
v := minlist(5, 7, 2, 9)
w := maxlist(CAP(1), CAP(2))
```

Inline "if" function

```
MAX_INVEN(t) := if(t  MAX_TIME, 1000, 0)

Inven(t) := stock(t) = buy(t) - sell(t) +
              if(t  1, stock(t-1), 0)
```

Matrix export to file

```
exportprob    ! Outputs LP/MIP portion handled by
! Mosel core; use solver-specific routines such as
! 'writeprob' of mmxprs for complete matrix output
```

File handling

```
fopen         fclose        fselect
getfid        getfname      getreadcnt
iseof         fflush        fskipline
fwrite[_] / fwriteln[_]
read / readln   write[_] / writeln[_]
```

String handling

```
strfmt        substr        _
```

Access and modify model objects

```
getcoeff[s]   setcoeff      getvars
gettype       settype       makesos1      makesos2
sethidden     ishidden      setname       setrange
getnbdim      getsize
getelt        getfirst      getlast       findfirst
findlast      gethead       gettail       cutelt
cutfirst      cutlast       cuthead       cuttail
reverse       getreverse    splithead     splittail
```

Access solution values

```
getobjval
getsol        getrcost
getslack      getact        getdual
```

Exit from a model

```
exit
```

Mosel controls	`getparam`	`setparam`	`localsetparam`	`restoreparam`
Date/time	`currentdate`	`currenttime`	`timestamp`	
Bit value handling	`bitflip`	`bitneg`	`bitset`	
	`bitshift`	`bittest`	`bitval`	
Handling unions	`geteltype`	`getstruct`	`gettypeid`	`isdefined`
Miscellaneous	`asproc`	`assert`	`compare`	`datablock`
	`dumpcallstack`	`memoryuse`	`newmuid`	
	`publish`	`unpublish`	`reset`	
	`setioerr`	`setmatherr`	`versionnum`	`versionstr`

- **Overloading of subroutines**
  - Some functions or procedures are *overloaded*: a single subroutine can be called with different types and numbers of arguments
- **Additional subroutines** are provided by *Mosel library modules*, which extend the basic Mosel language, *e.g.*,
  - *mmxprs*: Xpress Optimizer
  - *mmodbc*: ODBC data connection
  - *mmsheet*: accessing spreadsheets
  - *mmsystem*: system calls; text handling; date and time types
  - *mmjobs*: handling multiple models and (remote) Mosel instances
  - *mmsvg*: graphics

  ⇒ See the 'Mosel Language Reference Manual' for full details
- **User-defined functions and procedures**
  - You can also write your own functions and procedures within a Mosel model
  - Structure of subroutines is similar to a `model` (may have `declarations` blocks)
  - User subroutines may define overloaded versions of built in subroutines

  ⇒ See examples in Chapter *Functions and procedures*
- **Packages**
  - Additional subroutines may also be provided through *packages* (Mosel libraries written in the Mosel language as opposed to Mosel modules that are implemented in C)

  ⇒ See the Chapter *Packages* for further detail

A.7 Constraint handling

```
Ctr1:= 2*x + y = 10        ! Named constraints
Ctr2:= x is_integer

2*x + y = 10               ! Anonymous constraints
y = 5
```

Named constraints can be accessed:

```
val:= getact(Ctr)
getvars(Ctr, vars)
```

hidden:	`sethidden(Ctr, true)`
redefined:	`Ctr:= x+y = 10` `Ctr:= 2*x+5*y = 5`
modified:	`Ctr += 2*x` `settype(Ctr, CT_UNB)`
deleted (reset):	`Ctr:= 0`

Anonymous constraints are constraints that are specified without assigning them to a `linctr` variable. *Bounds* are (to Mosel) just simple constraints without a name. Anonymous constraints are applied in the optimization problem just like ordinary constraints. The only difference is that it is not possible to refer to them again, either to modify them, or to examine their solution value.

A.8 Problem handling

- Mosel can handle several *problems* in a given *model* file. A default problem is associated with every model.
- Built in type `mpproblem` to identify mathematical programming problems
  - The same decision variable (type `mpvar`) may be used in several problems
  - Constraints (type `linctr`) belong to the problem where they are defined
- The statement `with` allows to open a problem (= select the active problem):

```
declarations
  myprob: mpproblem
end-declarations
...
with myprob do
  x+y = 0
end-do
```

- Modules can define other specific problem types. New problem types can also be defined by combining existing ones, for instance:

```
mypbtyp = mpproblem and somepbtype
```

- Problem types support assignment: `P1:= P2`
  and additive assignment: `P1 += P2`

Appendix B

Good modeling practice with Mosel

The following recommendations for writing Mosel models establish some guidelines as to how to write "good" models with Mosel. By "good" we mean re-usability, readability, and perhaps most importantly, efficiency: when observing these guidelines you can expect to obtain the best possible performance of Mosel for the compilation and execution of your models.

B.1 Using constants and parameters

Many mathematical models start with a set of definitions like the following:

```
NT:= 3
Months:= {'Jan', 'Feb', 'Mar'}
MAXP:= 8.4
Filename= "mydata.dat"
```

If these values do not change later in the model, they should be defined as *constants*, allowing Mosel to handle them more efficiently:

```
declarations
 NT = 3
 Months = {'Jan', 'Feb', 'Mar'}
 MAXP = 8.4
 Filename= "mydata.dat"
end-declarations
```

If such constants may change with the model instance that is solved, their definition should be moved into the `parameters` block (notice that this possibility only applies to simple types, excluding sets or arrays):

```
parameters
 NT = 3
 MAXP = 8.4
 Filename = "mydata.dat"
end-parameters
```

Mosel interprets these parameters as constants, but their value may be changed at every execution of a model, *e.g.*

```
mosel exec mymodel NT=5 MAXP=7.5 Filename="mynewdata.dat"
```

B.2 Naming sets

It is customary in mathematical models to write index sets as 1, ..., N or the like. Instead of translating this directly into Mosel code like the following:

```
declarations
 x: array(1..N) of mpvar
end-declarations
```

```
sum(i in 1..N) x(i) >= 10
```

it is recommended to name index sets:

```
declarations
 RI = 1..N
 x: array(RI) of mpvar
end-declarations

sum(i in RI) x(i) >= 10
```

The same remark holds if several loops or operators use the same intermediate set(s). Instead of

```
forall(i in RI | isodd(i)) x(i) is_integer
forall(i in RI | isodd(i)) x(i) <= 5
sum(i in RI | isodd(i)) x(i) >= 10
```

which calculates the same intermediate set of odd numbers three times, it is more efficient to define this set explicitly and calculate it only once:

```
ODD:= union(i in RI | isodd(i)) {i}

forall(i in ODD) x(i) is_integer
forall(i in ODD) x(i) <= 5
sum(i in ODD) x(i) >= 10
```

Alternatively, loops of the same type and with the same index set(s) may be regrouped to reduce the number of times that the sets are calculated:

```
forall(i in RI | isodd(i)) do
 x(i) is_integer
 x(i) <= 5
end-do
sum(i in RI | isodd(i)) x(i) >= 10
```

B.3 Finalizing sets and dynamic arrays

The declaration of an array in Mosel has one of these two forms

1. Explicit declaration as sparse array by using one of the keywords `dynamic` or `hashmap`.
2. 'Standard' declaration, resulting in a dense array that is either *static* (all index sets are known) or *not fixed* (some or all indexing sets are unknown at the point where the declaration takes place).

If an array is used to represent dense data one should avoid defining it as a sparse array as that uses more memory and is slower than the corresponding dense array.

In many optimization models, dense arrays are created as non-fixed arrays because their contents is initially unknown—but there is no real need to treat them as dynamic structures throughout the whole model as they remain unchanged once they have been initialized.

The *automatic finalization* mechanism of Mosel therefore transforms such initially dynamic sets/non-fixed arrays as to handle them more efficiently. As an additional advantage, set finalization allows Mosel to check for 'out of range' errors that cannot be detected if the sets are allowed to grow dynamically.

- By default, `initializations` blocks finalize the sets they initialize and also the index sets of initialized dense arrays.
- Data of non-dynamic arrays is read before finalization of the index sets in order to create the arrays static.
- Arrays that are not explicitly declared as sparse arrays are only allocated when they are first

accessed: this allows these arrays to be static even if their index sets are finalized after the declaration of the arrays.

So, code like the following example

```
declarations
 S: set of string
 A,B: array(S) of real
 x: array(S) of mpvar
end-declarations

initializations from "mydata.dat"
 A
end-initializations

sum(s in S) B(s)*x(s)
```

where all arrays are declared as dense arrays that are not fixed (their size is not known at their declaration) but only `A` that is initialized using a data file really needs to be non-fixed, will be treated by Mosel as if you had written the following

```
declarations
 S: set of string
 A: array(S) of real
end-declarations

initializations from "mydata.dat"
 A
end-initializations

finalize(S)

declarations
 B: array(S) of real
 x: array(S) of mpvar
end-declarations
```

That is, `B` and `x` are created as static arrays, making the access to the array entries more efficient.

As a general rule, the following sequence of actions gives better results (in terms of memory consumption and efficiency):

1. Declare data arrays and sets that are to be initialized from external sources.
2. Perform initializations of data.
3. Finalize all related sets.
4. Declare any other arrays indexed by these sets (including decision variable arrays).

Note: there are several possibilities to stop Mosel from applying automatic finalization to model objects:

- Declare arrays explicitly as `dynamic` or `hashmap` arrays. (See examples in Sections 3.2 and 3.3.1.)
- Declare sets explicitly as `dynamic` in which case they cannot be finalized.
- Use control parameter `autofinal` to enable/disable automatic finalization *locally*:

```
setparam("autofinal", false)
initializations from "datafile.dat"
  ...
end-initializations
setparam("autofinal", true)
```

- Use option `noautofinal` to disable automatic finalization *globally* for the whole model:

```
model "modelname"
 options noautofinal
```

B.4 Ordering indices

Especially when working with sparse arrays, the sequence of their indices in loops should correspond as far as possible to the sequence given in their declaration. For example an array of variables declared by:

```
declarations
 A,B,C: range
 x: array(A,B,C) of mpvar
end-initializations
```

that is mostly used in expressions like `sum(b in B, c in C, a in A) x(a,b,c)` should preferrably be declared as

```
declarations
 A,B,C: range
 x: array(B,C,A) of mpvar
end-declarations
```

or alternatively the indices of the loops adapted to the order of indices of the variables.

B.5 Use of `exists`

The Mosel compiler is able to identify sparse loops and optimizes them automatically, such as in the following example:

```
declarations
 I=1..1000
 J=1..500
 A: dynamic array(I,J) of real
 x: array(I,J) of mpvar
end-declarations

initializations from "mydata.dat"
 A
end-initializations

C:= sum(i in I,j in J | exists(A(i,j))) A(i,j)*x(i,j) = 0
```

Notice that we obtain the same definition for the constraint `C` with the following variant of the code, but no loop optimization takes place:

```
C:= sum(i in I,j in J) A(i,j)*x(i,j) = 0
```

Here all index tuples are enumerated and the corresponding entries of `A` are set to 0. Similarly, if not all entries of `x` are defined, the missing entries are interpreted as 0 by the sum operator.

The following rules have to be observed for efficient use of the function `exists`, :

1. The arrays have to be indexed by named sets (here `I` and `J`):

```
A: dynamic array(I,J) of real              ! can be optimized
H: hashmap array(I,J) of real              ! can be optimized
B: dynamic array(1..1000,1..500) of real   ! cannot be optimized
```

2. The same sets have to be used in the loops:

```
forall(i in I,j in J | exists(A(i,j)))              ! fast
K:=I; forall(i in K,j in 1..500 | exists(A(i,j)))   ! slow
```

3. The order of the sets has to be respected, particularly for `dynamic` arrays:

```
forall(i in I,j in J | exists(A(i,j)))              ! fast
forall(j in J,i in I | exists(H(i,j)))              ! slower
forall(j in J,i in I | exists(A(i,j)))              ! slowest
```

4. The `exists` function calls have to be at the beginning of the condition:

```
forall(i in I,j in I | exists(A(i,j)) and i+j<>10)  ! fast
forall(i in J,j in J | i+j<>10 and exists(A(i,j)))  ! slow
```

5. The optimization does not apply to `or` conditions:

```
forall(i in I,j in J | exists(A(i,j)) and i+j<>10)  ! fast
forall(i in I,j in J | exists(A(i,j)) or i+j<>10)   ! slow
```

B.6 Structuring a model

Procedures and functions may be introduced to structure a model. For easy readability, the length of a subroutine should not exceed the length of one page (screen).

Large model files could even be split into several files (and combined using the `include` statement).

B.7 Transforming subroutines into user modules

The definitions of subroutines that are expensive in terms of execution time and are called very often (*e.g.* at every node of the Branch-and-Bound search) may be moved to a user module. Via the Mosel Native Interface it is possible to access and change all information in a Mosel model during its execution. See the Mosel Native Interface User Guide for a detailed description of how to define user modules.

B.8 Algorithm choice and parameter settings

The performance of the underlying solution algorithm has, strictly speaking, nothing to do with the efficiency of Mosel. But for completeness' sake the reader may be reminded that the subroutines `getparam` and `setparam` can be used to access and modify the current settings of parameters of Mosel and also those provided by modules, such as solvers.

The list of parameters defined by a module can be obtained with the Mosel command

```
exam -p module_name
```

With Xpress Optimizer (module *mmxprs*) you may try re-setting the following control parameters for the algorithm choice:

- LP: `XPRS_PRESOLVE`
- MIP: `XPRS_PREPROBING`, `XPRS_MIPPRESOLVE`, `XPRS_CUTSTRATEGY`, `XPRS_HEUREMPHASIS`, `XPRS_SBEFFORT`, `XPRS_NODESELECTION`
- Other useful parameters are the criteria for stopping the MIP search: `XPRS_MAXNODE`, `XPRS_MAXMIPSOL`, `XPRS_TIMELIMIT`, `XPRS_SOLTIMELIMIT`, the cutoff value (`XPRS_MIPADDCUTOFF`, `XPRS_MIPABSCUTOFF`), and various tolerance settings (*e.g.* `XPRS_MIPTOL`).

Refer to the Xpress Optimizer Reference Manual for more detail.

You may also add priorities or preferred branching directions with the procedure `setmipdir` (documented in the chapter on *mmxprs* in the Mosel Reference Manual).

Appendix C

Character encoding in Mosel

This chapter addresses a number of questions relating to character encoding, in particular:

- What is a 'character encoding', 'character map', 'code page'?
- What is Unicode?
- What is the meaning of UTF-8,16,32 and UCS-2?
- What is a BOM?
- Which character encoding is configured on my computer?
- Which files are concerned by character encoding in Mosel?
- How can I convert the character encoding of a text file?

C.1 What is a 'character encoding', 'character map', 'code page'?

Although these terms are not strictly equivalent they all relate to the same problematic: how to represent a symbol (or *character*) in a computer system. Such a representation is characterized by 2 properties:

1. a *character map* to associate each symbol to a unique numerical ID (or *code point*). For instance US-ASCII defines 128 positions to represent the letters, digits and punctuation commonly used in English: "exclamation mark" (!) has code point 33, "zero" has code point 48, "Capital Letter A" has code point 65, *etc.*
2. an *encoding method* to actually represent each code point in memory. With ASCII, 7 bits are sufficient to encode the entire code set: each character is usually encoded on a single byte.

Various character encodings have been invented to satisfy local requirements around the world. For example, ISO-8859-1 is an 8-bit extension of ASCII (*i.e.* the 128 first code points of this encoding are the same as ASCII) specifically designed for a group of European languages: it adds a set of accented letters to standard ASCII. Another version, ISO-8859-7 is suitable for Greek but cannot represent accented letters such as those used in French.
When the number of code points exceeds 256 it is required to switch to a *multi-byte encoding*. Shift-JIS (used in Japan) is an example of multi-byte encoding: each character is encoded using either 1 or 2 bytes.

Typically a computer system is set up with some national encoding suitable to handle the symbols required by the local language. For instance a Windows system installed in Germany uses encoding CP1252 (where CP stands for *Code page*) that supports symbols like 'ß' or 'ö' but will not be able to display any Greek (*e.g.* ' θ ') or Hebrew characters (*e.g.* ' א ').

C.2 What is Unicode?

Unicode is a universal encoding aimed at representing all known symbols such that a single encoding can be used for any country/language. Unicode is widely adopted and most computer systems use it internally to store character strings: the Windows operating system (and file system) uses this encoding as well as most Unix/Linux systems. Programming environments like Java or .NET are also based on Unicode.

Note that in China the GB18030 encoding is preferred to Unicode: this is a universal encoding published by the Chinese National Standard.

C.3 What is the meaning of UTF-8,16,32 and UCS-2?

Unicode defines the mapping between code points and symbols, the effective encoding is specified by a *Unicode Transformation Format (UTF)*. The most commonly used UTF encodings are:

UTF-32 a character is represented by a 4-byte integer

UTF-16 a character is represented by 1 or 2 2-byte integers

UTF-8 a character requires between 1 and 4 bytes

Compared to the other UTF encodings UTF-8 has the advantage of being compatible with ASCII: a text that consists only of ASCII characters has the same representation in UTF-8 and ASCII. As a consequence UTF-8 is also more compact than the other UTF encodings for English and most European languages (because the majority of symbols are included in the ASCII set).

UCS-2 (Universal Character Set v2) is a deprecated encoding originally used in Windows and Java: it encodes each character on a 2-bytes integer and is therefore limited to the first 65536 code points of Unicode, this is why it has gradually been replaced by plain UTF-16.

C.4 What is a BOM?

For UTF-16 and UTF-32 the byte ordering has to be known (in fact we should refer to UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE to take into account the endianness of the encoding). In order to avoid incorrect interpretation of these encodings a *Byte Order Mark (BOM)* may be put at the beginning of documents: it consists in a sequence of bytes that identifies both the encoding (UTF-16 or UTF-32) and the byte ordering used (Little Endian or Big Endian).

Although UTF-8 does not require any byte ordering information, a dedicated BOM can be used with this encoding: its primary purpose is to differentiate UTF-8 from other byte-oriented encodings. If not interpreted this marker takes the form of the 3-characters sequence ”ï»¿” (in ISO-8859-1 or CP1252): a document starting with this sequence must be read with an UTF-8 enabled software.

C.5 Which character encoding is configured on my computer?

You can use the command ’xprnls info’ of the XPRNLS command tool to identify which encoding is used on your system. The following example shows the output produced for western European Windows / ’latin’ encoding with UK English as the selected language (the program output is highlighted in bold face):

```
>xprnls info
Language: en
Default encodings:
 System:     CP1252
 Console:    CP437
 File names: CP1252
 Wide chars: UTF-16LE
```

Note that Xpress Workbench works with UTF-8 character encoding, independent of the system settings.

C.6 Which files are concerned by character encoding in Mosel?

Starting with version 4.0 Mosel is working in UTF-8. This concerns

- the internal representation of text
- all external APIs (*i.e.* all Mosel libraries)
- the communication with the system via Unicode (Windows) or system encoding (Posix)

All streams and text files default to UTF-8. There is no impact on applications that only use pure ASCII (first 127 characters), but *text data files and source code* using other encodings might require conversions or tagging. Note that no changes are required for other file types such as spreadsheets or databases.

Model source and text data files in Mosel format: Specify the encoding with the annotation `!@encoding`. For example if you are editing your model with an editor that employs the encoding CP1252:

```
!@encoding CP1252
model "my testmodel"
 ...
```

Other text/string input or output: Convert the encoding via the `enc:` prefix to file names and streams or by using the conversion routines of the XPRNLS library or command tool (see paragraph 'How can I convert the character encoding of a text file' below).

C.7 How can I convert the character encoding of a text file?

Text format data files (other than the Mosel `initializations` format for which the `!@encoding` marker can be used) such as CSV files or files accessed via `fopen` that do not use UTF-8 encoding need to be converted with the `'enc:'` prefix when accessing them from within a Mosel model.
Example:

```
! Encoding names are operating system dependent, eg CP1252, ISO88591
fopen(enc:GB18030,testdata.txt", F_INPUT)

! Apply an encoding along with a BOM to a text file, saving it with a new name
fcopy("myfile.csv", F_INPUT, "enc:UTF-8+bom,mynewfile.csv", F_OUTPUT)
```

It is usually preferrable to specify the encoding used by a data file as shown above, but Mosel also implements shorthands for encodings configured on the system running the model.

```
! Encoding aliases:
! raw, sys, wchar, fname, tty, ttyin, stdin, stdout, stderr
initializations to "mmsheet.csv:enc:sys,testoutput.csv"
  ...
end-initializations
```

Using the prefix `enc:sys` means that the default system encoding is employed (which corresponds to the behaviour of Mosel versions prior to Mosel 4).

On the API level, you can use the *XPRNLS library* to convert to/from UTF-8 encoding (please see the reference manual *XPRNLS command tool and library* for the full documentation of its functionality):

- this library is platform independent and has no external dependency
- it handles encoding conversions between UTF-8 and local encodings
- it implements UTF-8/16/32(LE+BE), ISO-8859-1/15, ASCII, CP1252
- other supported encodings depend on the operating system

```
// Open a file using the C function 'fopen' with a file name coming from Mosel
f = fopen(XNLSconvstrto(XNLS_ENC_SYS,filename,-1,NULL),"r");
```

Alternatively, you can use the *XPRNLS command tool* for converting the character encoding of text files between any two supported encodings:

```
xprnls conv -f CP1252 -t UTF8 -o outfile.txt myfile.txt
```

Note: you can display the list of the available *xprnls* commands by entering

```
xprnls
```

at the command prompt.

Appendix D

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.

Index

Symbols
*, 7, 63
+, 15, 63, 67
+=, 64, 67
, , 15
–, 15, 63, 67
–=, 64, 67
–>, 77, 85
. . . , 81
::, 13, 241
<=, 7
=, 7
>=, 7

A
abs, 34, 84
addcuts, 98
alias, 77
and, 45
annotation, 220
    character encoding, 256
    deprecation, 183
    version compatibility, 183
annotation category, 220
annotation name, 220
any, 45, 74, 242
application
    access model, 117
    compile model, 115
    data exchange, 120, 136, 147, 157
    execute model, 116
    model parameters, 117
    solution access, 117
array, 12, 57
    access functions, 59
    automatic, 61
    declaration, 12
    dense, 25, 119, 122, 135, 136, 146, 147, 157, 172, 249
    dynamic, 25, 27, 95, 102, 172
    finalize, 249
    index set, 59
    initialization, 13, 17
    initialization operator, 13
    input data format, 25
    iterator, 177
    multi-dimensional, 12, 13, 25
    non-fixed, 25
    sparse, 119, 123, 135, 137, 146, 148, 157, 158, 172
    static, 27, 172, 249
array, 45, 57, 61, 91
as, 45
as counter, 53
automatic array, 61
automatic finalization, 27, 62, 249
    disable, 250

B
baseline scenario, 204
BIM file, 115
bin, 188
binary format, 188
binary model file, 195
binary variable, 32
blending constraint, 16
BOM, *see* Byte Order Mark
boolean, 45, 57
bounded variable, 16
BRE, 218
break, 45, 56
Byte Order Mark, 255

C
C interface, 115
callback, 99
case, 45
cb, 124, 188
ceil, 101
character encoding
    compatibility, 256
    conversion, 256
character map, 254
code page, 254
column, *see* variable
column generation, 100
combining solvers, 207
comment, 7
    multiple lines, 7
comparison
    list, 67
    set, 64
comparison tolerance, 98
compile, 8, 115
    to memory, 195
compile, 195
condition, 25, 50, 252
conditional generation, 26
conditional loop, 53
connect, 197
constant, 12
constant, 45, 214
constant list, 65
constant set, 62
constant type, 59

constraint
general, 34
hide, 111
indicator, 34
MVLB, 96
named, 13
non-negativity, 6, 7
type, 110
Constraint Programming, 207
continuation line, 15
count, 45, 54
counter, 45
counters, 53
CP, *see* Constraint Programming
create, 26, 59, 74
cross-recursion, 82
csv, 193
CSV format, 21
cut generation, 94
cut manager, 98
cut pool, 98
cutting plane method, 94
cutting stock problem, 100

D
data
communication, 120, 136, 147
declaration, 172
dense, 172
exchange with application, 120, 136, 147, 157
initialization, 172
input from database, 18
input from file, 17, 25, 28
multi-dimensional array, 25, 28
output, 89
sparse, 172
sparse format, 28
data file, 187
format, 17, 25
data format
dense, 122, 136, 147, 157
sparse, 123, 137, 148, 158
database, 18
date, 44
date, 212, 214
datefmt, 213
datetime, 212, 215
debug, 166
debugger, 166
debugging, 46
decision variable, *see* variable, 5
array, 13
declaration
array, 12
public, 179
subroutine, 82
declarations, 7, 45, 80
decomposition, 194
default I/O driver, 187
deflate, 194
delcell, 58
delete temporary files
model, 133, 143, 155
dense, 119, 135, 146
dense array, 25, 58
dense data, 60, 172, 249
dense data format, 122
dense format, 136, 147, 157
deviation variable, 110
difference, 63, 64
Dim, 147
diskdata, 29, 90, 91, 191
distributed computing, 197
div, 45
do, 45
doc, 221
dotnet, 150, 153, 189
dotnetraw, 147, 148, 190
dynamic, 25, 45, 58, 172, 249
dynamic array, 25, 27, 58, 95, 172
dynamic data input, 126, 139, 150
dynamic list, 65
dynamic output retrieval, 125, 139, 150
dynamic set, 62

E
efficiency, 169
elif, 45
else, 45
embedding
data exchange, 120, 136, 147, 157
model access, 117
enc:, 256
encoding, 256
encoding method, 254
end, 45
end-declarations, 7
end-do, 52
end-function, 79
end-initializations, 17
end-model, 7
end-procedure, 79
enumeration
dense array, 119, 135, 146
inverse order, 66
set, 118, 134, 145, 156
sparse array, 120, 136, 147
ERE, 218
error
data, 46
logical, 46
redirection, 129, 142, 153, 159, 162
run time, 47
syntax, 46
error handling, 29
error stream, 162
redirecting, 31
ETC_SPARSE, 90, 91
ETC_OUT, 89
ETC_SPARSE, 89

evaluation, 45, 91
exam, 43, 188
excel, 193
Excel spreadsheet, 21
execute, 8
execution speed, 169
exists, 25, 173, 251, 252
exportprob, 27, 187
extended file name, 187

F
F_APPEND, 89
F_OUTPUT, 89
false, 45
fclose, 31, 89, 187
fdelete, 196
feasibility tolerance, 98
field, 57
    access, 70
file
    generalized, 187
file handling, 187
file output, 89
    solution, 91
finalized, 27
find_identifier, 156, 157
FindIdentifier, 145
findIdentifier, 134
finish, 116, 120
FirstIndex, 145
fixed set, 62
fixed size array, 58
flow control, 50
fmax, 34
fmin, 34
fopen, 31, 89, 187
forall, 14, 25, 45, 52–54, 56
forall–do, 52
format
    date, 213
    real number output, 92
    text output, 87
    time, 213
formattext, 88
forward, 45, 82, 98
free variable, 107
from, 45
fully qualified entity name, 181
function, 79, 252
function, 45, 79

G
general constraint, 34
generalized file, 187
getFirstTEIndex, 136
getannotations, 221
getasnumber, 214, 215
getcoeff, 84
getDimension, 136
geteltype, 75
getexitcode, 195
getFirstIndex, 134, 135
getfstat, 30
getindex, 59
getLastIndex, 134
getnbdim, 59
getobjval, 8
getparam, 30, 180
getreadcnt, 30
getreverse, 66
getsize, 59, 64
getsol, 8, 37, 84
getstruct, 75
gettype, 110
gettypeid, 74
getvalue, 195
global annotation, 220
gnuplot, 193
Goal Programming, 109
    Archimedian, 109
    lexicographic, 109
    pre-emptive, 109
graphics, 198
gzip, 194

H
hashmap, 45, 58, 172, 249
hashmap array, 58
head, 67
hide
    constraint, 111
hybrid solution approaches, 207

I
I/O driver, 91, 187
    bin, 188, 189
    cb, 188
    csv, 193
    default, 187
    deflate, 194
    diskdata, 191
    dotnet, 189
    dotnetraw, 190
    excel, 193
    gzip, 194
    java, 191
    jraw, 191
    mem, 188
    mempipe, 192
    null, 189
    oci, 192
    odbc, 193
    pipe, 193
    raw, 189
    rcmd, 192
    rmt, 192
    shmem, 192
    sysfd, 189
    text, 193
    url, 191

xls, 193
xlsx, 193
xsrv, 192
xssh, 192
I/O error, 29
if, 45, 50, 53
if-then, 50
if-then-else, 55
implicitly dynamic array, 249
imports, 45, 174
in, 45, 64
include, 45, 174, 252
index
multiple, 53
index set, 12, 15
index set type, 59
indicator, 34
indicator constraint, 34
Indices, 146
info, 172
initialisations, 45
initialization
array, 13, 17
list, 65
set, 61
initializations, 17, 28, 45, 89, 137, 148, 187, 196, 249
initializations from, 17
initializations to, 182
Insight, *see* Xpress Insight
integer, 45, 57
integer knapsack problem, 103
Integer Programming, 37
integer variable, 32
inter, 45
interrupt
loop, 56
intersection, 63
ioctrl, 30
iostatus, 30
IP, *see* Integer Programming
is, 45, 74, 75
is_binary, 45
is_continuous, 45
is_free, 45
is_integer, 45
is_partint, 45
is_semcont, 45
is_semint, 45
is_sos1, 33, 37, 45
is_sos2, 33, 40, 45
is_binary, 32
is_integer, 32
is_partint, 33
is_semcont, 33
is_semint, 33
isdefined, 85
isqueueempty, 197
isvalid, 214
iterator, 177

J
java, 138, 142, 191
jraw, 136, 137, 191

K
kalis, 210
knapsack problem, 11
integer, 103

L
largest common divisor, 55
LastIndex, 145
limit, *see* bound
linctr, 45, 57
line break, 15
Linear Programming, 4, 37
Linear Programming problem, 6
list, 57
comparison, 67
concatenation, 67
constant, 65
dynamic, 65
enumeration, 66
initialization, 65
merging, 67
operators, 67
list, 45, 57
load, 8, 115
load, 195
loadprob, 130
local declaration, 54
loop, 14, 50, 52
conditional, 53, 173
interrupting, 56
nested, 56
sparse, 251
LP, *see* Linear Programming
lsmods, 172
lssymb, 172

M
makesos2, 40
match
regular expression, 218
Mathematical Programming, 4
max, 45, 53
maximize, 130
maximum, 51
mc, 221
mem, 122, 188
memory consumption, 169
mempipe, 192
meta data, 220
min, 45
minimum, 51
MIP, *see* Mixed Integer Programming
MIQP, *see* Mixed Integer Quadratic Programming
Mixed Integer Programming, 4, 32, 94
Mixed Integer Quadratic Programming, 207
mmdotnet, 189

mmetc, 29, 90, 191
mmhttp, 191
mminsight, 198
mmjava, 191
mmjobs, 192, 194
mmnl, 207
mmoci, 192
mmodbc, 18, 193
mmreflect, 177
mmrobust, 207
mmsheet, 18, 21, 193
mmsvg, 198
mmsystem, 44, 193
mmxml, 199, 200
mmxnlp, 34, 209
mmxprs, 8, 44, 47, 130
mod, 45
model, 6
  access from application, 117
  compile, 115
  coordination, 195
  data from application, 120, 136, 147, 157
  execute, 8, 116
  parameters, 117
  reset, 116, 120, 132, 143
  run, 8
  unload, 116
model, 7, 45, 174, 175
model documentation, 223
model file, 115
model structure, 252
modeling
  efficiency, 169
module, 43, 187
  I/O driver, 189
monthnames, 213
Mosel Remote Launcher, 192
MOSEL_DSO, 174
moseldoc, 223
MoselPy, 155
moselpy:, 157
MP, *see* Mathematical Programming
mpproblem, 45, 104
mpvar, 7, 13, 25, 45, 57
multi-objective optimization, 109, 111
multiple indices, 53
multiple models, 194
multiple nodes, 197
multiple problems, 103, 104
MVLB constraint, 96

N
name
  constraint, 13
namespace, 45, 181
namespace group, 181
nbread, 28, 30
negation, 64
nested loops, 56
next, 45
nextTEIndex, 136
nextIndex, 135
nlctr, 34
NLP, *see* Non-linear Programming
noautofinal, 250
noindex, 122
Non-linear Programming, 207
non-negative variable, 6, 7
non-negativity constraint, 6, 7
not, 45, 64
nsgroup, 45, 181
nssearch, 45, 181
null, 31, 189
number output format, 92

O
objconfig, 111
objecive function
  multiple, 111
objective function, 6, 7
oci, 192
ODBC, 18
odbc, 193
of, 45
open, 189
operator
  counter, 54
  set, 64
optimization, 8
options, 45
optnode callback, 99
or, 45, 252
output, 8
  disable, 189
  file, 89
  formatted, 110
  formatting, 87
  redirection, 129, 142, 153, 159, 162, 189
  splitting, 189
output file, 162
overloading, 83

P
package, 174
  internal name, 174
  location, 174
  name, 174
  version management, 183
package, 45, 174, 175
package parameter, 179
parallel solving, 194
parameter, 17
  comparison tolerance, 98
  global, 80
  local, 80
  number output format, 92
  subroutine, 80
parameters, 117, 122, 136, 147
parameters, 18, 45, 248
parser, 46

partial integer variable, 33
perfect number, 52
piecewise linear, 40
    segment, 42
pipe, 193
prime number, 64, 117, 133, 144, 156
problem
    decomposition, 194
    multiple, 103, 104
    solving, 8
procedure, 79, 252
procedure, 45, 79
prod, 45
profile, 169
profiler, 169
project planning problem, 35
public, 27, 45, 70, 100, 117, 133, 156, 179, 204
pwlin, 34, 41
pws, 42
Python interface, 155

Q
QCQP, *see* Quadratically Constrained Quadratic Programming
QP, *see* Quadratic Programming
qsort, 82, 215
Quadratic Programming, 4, 207
Quadratically Constrained Quadratic Programming, 207
quick sort, 82
quit, 166

R
range, 45, 52
range set, 12
raw, 123, 189
rcmd, 192
read, 28
readcnt, 30
readln, 28
real
    output format, 92
real, 45, 57
REALFMT, 92
record, 57
record, 45, 57, 69
recursion, 82, 106
redirecting output, 189
reference row entries, 33
reference to operator, 77, 85
reflection, 74
REG_EXTENDED, 218
REG_ICASE, 218
regex, *see* regular expression
regular expression, 218
    matching, 218
relocating documentation, 226
remote execution, 197
repeat, 45
repeat–until, 55, 56
requirements, 45
reset
    model, 116, 120, 132, 143
reset, 58, 85, 155
return, 45
returned, 79
reverse, 66
rmt, 192
Robust Optimization, 207
row, *see* constraint
run, 8, 115
run, 155, 195

S
scalar
    I/O in memory, 127, 141, 152, 159
segment
    piecewise linear, 42
selection statements, 50
semi-continuous integer variable, 33
semi-continuous variable, 33
set, 57, 117, 133, 144, 156
    comparison, 64
    constant, 27, 62
    dynamic, 62
    finalize, 249
    finalized, 27
    fixed, 62
    initialization, 61
    maximum, 51
    minimum, 51
    string indices, 15
    type, 57
set, 45, 57
set of constants, 214
set of strings, 15
set operation, 63
set operator, 64
sethidden, 103, 111
setparam, 30, 180
shared, 45
shell sort, 56
shmem, 192
sleep, 168
SLP, 207
solution output, 91
solution value, 118, 134, 145, 156
solvers
    combining, 207
solving, 8
sorting
    date, 215
    time, 215
sorting algorithm, 44, 56, 82
sparse, 25, 28, 90
    loop, 251
sparse array, 58, 172
sparse data, 60, 172
sparse format, 119, 123, 135, 137, 146, 148, 157, 158
sparsity, 23

Special Ordered Set of type one, 33, 37
Special Ordered Set of type two, 33
spreadsheet, 18
static array, 172
stop, 196
strfmt, 87
string, 45, 57
submodel, 194
  coordination, 195
  interaction, 195
  status, 195, 229
subproblem, 104, 111
subroutine, 79, 252
  declaration, 82
  definition, 82
  overloading, 83
  parameter, 80
  variable number of arguments, 81
subroutine reference, 85
subscript, 12
subset, 64
Successive Linear Programming, 106
sum, 45, 53, 67
summation, 13
superset, 64
syntax
  regular expression, 218
syntax error, 46
sysfd, 189
system call, 44

T
table, *see* array
tail, 67
tee, 189
TEIndices, 147
temporary directory, 189
temporary files
  delete, 133, 143
termination, 116, 120
text, 31, 193, 217
textarea, 219
textfmt, 88
then, 45
time, 212, 215
time measurement, 44
timefmt, 213
tmp, 189
to, 45
tolerance
  comparison, 98
  feasibility, 98
  real number output, 92
transport problem, 23, 119, 135, 146, 157
true, 45
type
  array, 57
  basic, 57
  constant, 12
  constraint, 110
  elementary, 57
  external, 57
  list, 57
  MP, 57
  record, 57
  set, 57
  structured, 57
  user, 72
type constructor, 73

U
unbounded variable, 107
Unicode, 255
Unicode Transformation Format, 255
union, 63, 74
union, 45, 53
union type, 177
unload
  model, 116
until, 45
url, 191
uses, 19, 45, 174
UTF, *see* Unicode Transformation Format

V
variable, 5
  binary, 14, 32
  bounds, 16
  conditional creation, 26
  free, 107
  integer, 14, 32
  lower bound, 7
  non-negative, 6, 7
  partial integer, 33
  semi-continuous, 33
  semi-continuous integer, 33
  unbounded, 107
VDL, *see* View Definition Language
version, 45, 183
version compatibility, 183
View Definition Language, 206

W
wait, 195
warning, 47
while, 45, 55, 56, 64
while-do, 55
with, 45, 54, 105
Workbench, 8
workdir, 30
write, 8, 87, 90
writeln, 8, 28, 87, 89, 90
writeprob, 27

X
xls, 193
xlsx, 193
XML document, 200
XML path, 200
xmldoc, 200
XPRDstoprunmod, 231

Xpress Global, 207
Xpress Insight, *see* Xpress Insight, 198
Xpress Kalis, 207
Xpress Optimizer, 130
Xpress Workbench, *see* Workbench
XPRM, 132, 143
XPRM.Init(), 143
XPRM_F_ERROR, 162
XPRM_F_OUTPUT, 162
XPRMcompmod, 187
XPRMexecmod, 116
XPRMexecmod, 117
XPRMfinddso, 130
XPRMfindident, 118
XPRMfinish, 116, 117
XPRMgetfirstarrtruentry, 120
XPRMgetnextarrtruentry, 120
XPRMgetarrdim, 120
XPRMgetarrsets, 120
XPRMgetelsetval, 118
XPRMgetfirstarrentry, 119
XPRMgetfirstsetndx, 118
XPRMgetlastsetndx, 118
XPRMgetnextarrentry, 119
XPRMloadmod, 116, 187
XPRMresetmod, 116, 117, 120
XPRMrunmod, 116, 187
XPRMsetdefstream, 129
xprmsrv, 192
XPRMunloadmod, 116, 117
xprnls, 255, 257
XPRNLS command tool, 255, 257
XPRNLS library, 257
XPRS_NLPSOLVER, 210
XPRS_PROBLEM, 130
XPRS_LOADNAMES, 47
XPRS_VERBOSE, 47
xsrv, 192
xssh, 192

Z

ZEROTOL, 98
zlib, 194