Xpress-Mosel

User Guide

How to Contact Dash

If you have any questions or comments on the use of Xpress[MP], please contact Dash technical support at:

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 313 5297
Fax: (201) 313 5299
e-mail: support-usa@dashoptimization.com

Elsewhere
Dash Optimization Limited
Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5TH
UK
Telephone: +44 1926 315862
Fax: +44 1926 315854
e-mail: support@dashoptimization.com

If you have any sales questions or wish to order Xpress[MP] software, please contact your local sales office or Dash sales at:

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 313 5297
Fax: (201) 313 5299
e-mail: sales@dashoptimization.com

Elsewhere
Dash Optimization Limited
Blisworth House, Church Lane
Blisworth
Northants NN7 3BX
UK
Telephone: +44 1604 858993
Fax: +44 1604 858147
e-mail: sales@dashoptimization.com

For the latest news and Xpress[MP] software updates please visit the Dash website at:
http://www.dashoptimization.com

Contents

Introduction - Chapter 1: Mosel

1.1 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 don't require pre-sizing. For instance, you don't 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 by Dash, including the Xpress^{MP} 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 sub problems 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.

1.2 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 linear inequalities. For example:

$$x + y \leq 6$$

Experience of a basic course in Mathematical or Linear Programming is recommended but is not essential.

For all but the simplest models you should also be familiar with the idea of summing over a range of variables. For example, if x_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:

$$\sum_{j=1}^{N} x_j$$

which says "sum the output from each production line x_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:

$$\sum_{j=1}^{N} x_j \geq 1000$$

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 Mosel should be straightforward.

1.3 Symbols and Conventions

We have used the following conventions within this guide:

- Square brackets [...] contain optional material.
- Curly brackets {...} contain optional material, one of which must be chosen.
- Entities in *italics* which appear in expressions stand for meta-variables. The description following the meta-variable always describes how it is to be used.
- The vertical bar symbol | is found on many keyboards as ¦, but often confusingly displays on-screen without the small gap in the middle. 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 decimal.
- Examples of commands, models and their output are printed in a `Courier font`. Filenames are given in lower case Courier.
- Mathematical objects are presented in *italics*.

1.4 The Structure of this Guide

This User Guide is structured into these main parts

Part A 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 B is designed to help those users who want to use the powerful programming languages 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.

Part C 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.

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**.**

Part A - Chapter 2: Getting Started with Mosel

A2.1 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 (don't use a word processor as an editor as this may not produce an ASCII file). 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.

A2.2 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 ten 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 larger chess set yields a profit of £20 and each smaller chess set a profit of £5. The problem is to decide how many sets of each kind should be made each week to maximize profit.

A2.2.1 A First Formulation

The joinery can *vary* the number of large and small chess sets produced: there are thus two *variables* in our model. We shall give these variables 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. Since the values of *small* and *large* are to be determined by optimization these are known as the *decision variables*, or more simply the *variables*.

The values which *small* and *large* can take may be *constrained* to be equal to, less than or greater than some constant. The joinery has a maximum of 400 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 $3\,small + 2\,large$. One *constraint* is thus:

$3\,small + 2\,large \leq 400$ (machine time)

which restricts the allowable combinations of small and large chess sets to those that do not exceed the man-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:

$small + 3\,large \leq 200$ (wood)

The joinery cannot produce a negative number of chess sets so two further constraints are:

$small \geq 0$
$large \geq 0$

The *objective function* is a linear function which is to be optimized, that is, maximized or minimized. It will involve some or all 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 aim of the joinery is to maximize profit. Since a large set contributes £20 to total profit while a small set contributes £5, the objective function is:

$profit = 5\ small + 20\ large$

The collection of variables, constraints and objective function that define our linear programming problem is called a *model*.

A2.3 Solving the Chess Set Problem

A2.3.1 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
  large: mpvar
 end-declarations

 profit:=  5*small + 20*large
 mc_time:= 3*small + 2*large <= 400
 wood:=      small + 3*large <= 200

end-model
```

Indentations are purely for clarity.

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 `mc_time` and `wood` 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 mathematical programming 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:

```
mc_time:= 3*small + 2*large <= 400
```

The name of the constraint is `mc_time`. 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.

A2.3.2 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^{MP}'s 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 chess2
 uses "mmxprs"
 declarations
  small: mpvar
  large: mpvar
 end-declarations

 profit:=  5*small + 20*large
 mc_time:= 3*small + 2*large <= 400
 wood:=      small + 3*large <= 200

 maximize(profit)

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

The line

```
uses "mmxprs"
```

tells Mosel that the Xpress Optimizer will be used to solve the LP. We need to tell Mosel that we shall be using the XPRS module, which 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.

`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.

A2.3.3 Running Mosel from a Command Line

When you have entered the complete model into a file (let's say it is called `chess2.mos`), we can proceed to get the solution to our problem. Three stages are required:

1 Compiling `chess.mos` to a compiled file, `chess.bim`

2 Loading the compiled file `chess2.bim`

3 Running the model we have just loaded.

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

```
C:\mosel\book>mosel

compile chess2
load chess2
run
quit
```

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

```
C:\mosel\book>mosel
** Mosel **
 (c) Copyright Dash Associates 1998-2001
>compile chess2
Compiling 'chess2'...
>load chess2
>run
small is 0
large is 66.6667
Best profit is 1333.33
Returned value: 0
>quit
Exiting..
```

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

```
cl chess2
run
quit
```

or even

```
cl chess2
ru
q
```

Another nice way to do all the steps from the command line is

```
C:\mosel\book>mosel -c "cl chess2; ru"
```

The `-c` option is followed by a list of commands enclosed in double quotes. Even nicer is to use this with Mosel's silent (`-s`) option:

```
C:\mosel\book>mosel -s -c "cl chess2; ru"
```

When the only output is

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

Part A - Chapter 3: Some Illustrative Examples

A3.1 The Burglar Problem

This chapter develops the basics of modeling set out in Chapter 2. It presents some further examples of the use of Mosel and introduces new features.

The first of these is the use of subscripts. Almost all models of any size have subscripted variables. Consider a model which represents the problem faced by a burglar. He sees 8 items, of different worths and weights. He wants to take the items of greatest total worth whose total weight is not more than the maximum he can carry. This is an example of a Knapsack Problem.

```
model Burglar
 uses "mmxprs"
 declarations
  Items = 1..8                   ! Index range for items
  VALUE: array(Items) of real    ! Value of items
  WEIGHT: array(Items) of real   ! Weight of items
  WTMAX=102                      ! Max weight allowed
  x: 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)*x(i)

 ! Weight restriction
  WtMax:= sum(i in Items) WEIGHT(i)*x(i) <= WTMAX

 ! All x are 0/1
  forall(i in Items) x(i) is_binary

 maximize(MaxVal)            ! Solve the MIP-problem

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

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

A3.1.1 Index Ranges

The line

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

introduces an index range, that is, a range of values over which an index will later range.

A3.1.2 Arrays

```
VALUE: array(Items) of real    ! Value of items
```

declares VALUE to a one-dimensional array of real values. In Mosel, reals are double precision. The other numeric type is integer. An individual element of the array is indexed by a value in the index 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 are declared likewise:

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

declares an array of decision variables x(1), x(2), x(3), …, x(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)
```

A3.1.3 Constants

```
WTMAX=102                      ! Max weight allowed
```

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.

A3.1.4 Assigning Values to Arrays

```
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.

A3.1.5 Summations

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

defines a linear expression called MaxVal as the sum $\sum_{i=1}^{8} VALUE_i * x_i$

A3.1.6 Making a Variable a Binary Variable

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
```

A3.1.7 Simple Looping

The statement

```
forall(i in Items) x(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 `x(1)`, `x(2)`, … , `x(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.

A3.1.8 Comments

The symbol `!` signifies the start of a comment, which continues to the end of the line. Longer comments can be written thus

```
(! Mosel example problem
   ====================
   file xxxx.yyy
   Example of the use of the Mosel language
   (c) yyyy Dash Associates
!)
```

where `(!` denotes the start of the comment, and `!)` the end.

A3.2 The Burglar Problem Revisited

Consider this model:

```
model Burglar2
uses "mmxprs"
declarations
 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
  WTMAX=102                      ! Max weight allowed
  x: array(Items) of mpvar
           ! 1 if we take item i; 0 otherwise
 end-declarations

 !   Item:  ca   ne  va  pi  tv  vi  ch  br
  VALUE := [15, 100, 90, 60, 40, 15, 10,  1]
  WEIGHT:= [ 2,  20, 20, 30, 40, 30, 60, 10]

 MaxVal:= sum(i in Items) VALUE(i)*x(i)
 ! Objective: maximize total value

 ! Weight restriction
  WtMax:= sum(i in Items) WEIGHT(i)*x(i) <= WTMAX

 ! All x are 0/1
  forall(i in Items) x(i) is_binary

 maximize(MaxVal)            ! Solve the MIP-problem

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

What have we changed? The answer is, “not very much”

A3.2.1 String Indices

The lines

```
Items={"camera", "necklace", "vase", "picture", "tv", "video",
  "chest", "brick"}           ! Index set for items
```

declare that this time `Items` is a set of indices (an Index Set) with the indices taking the string values `"camera"`, `"necklace"` etc.

If we run the model, we get

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

A3.2.2 Continuation Lines

Notice that the statement

```
Items={"camera", "necklace", "vase", "picture", "tv", "video",
  "chest", "brick"}           ! Index set for items
```

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)
```

A3.2.3 Correcting Syntax Errors

The parser of Mosel is able to detect a large number of errors that may occur when writing a model. In this section we shall try to analyze and correct some of these. If we compile the model

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

 profit=   5*small + 20*large
 mc_time:= 3*small + 2*large <= 400
 wood:=      small + 3*large <= 200

 maximize(profit)

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

we get the following output:

```
Mosel: E-100 at (1,7) of `error.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 (7,8) of `error.mos': Syntax error before `='.
Mosel: W-121 at (7,28) of `error.mos': Statement with no effect.
Mosel: E-100 at (11,16) of `error.mos': Syntax error before `profit'.
Mosel: E-100 at (13,39) of `error.mos': Syntax error.
Parsing failed.
```

There is a problem with the sign = in the 7th line:

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

In the model body the equality sign = may only be used in the definition of constraints. Constraints are linear relations between variables, but profit has not been defined as a variable, the parser therefore 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 `:=` .

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-100 at (11,17) of `error.mos': Syntax error.
Mosel: E-100 at (13,37) of `error.mos': Syntax error.
```

There is still a problem in line 11, 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 this line. 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.

We now have a closer look at line 13 (that has now become line 14 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 brackets. After adding the closing bracket 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
```

Besides the detection of syntax errors, Mosel may also give some help in finding run time errors. Going into details would lead too far at this place. 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.

A3.3 A Blending Example

This example illustrates how data may be read into tables from text files.

A3.3.1 The Model Background

A mining company has two types of ore available: Ore 1 and Ore 2. Denote the amounts of these ores to be used by x_1 and x_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 4.0 and 5.0. It sells for £125 per ton. The costs of the two ores vary, as do their availabilities.

Maximizing net profit (i.e., sales revenue less cost of raw material) gives us the objective function:

$$NET\_PROFIT = \sum_{j=1}^{2} (125 - COST_j) x_j$$

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:

$$\frac{\sum_{j=1}^{2} GRADE_j x_j}{\sum_{j=1}^{2} x_j}$$

This must be greater than or equal to 4.0 so, cross-multiplying and collecting terms, we have the constraint:

$$\sum_{j=1}^{2} (GRADE_j - 4.0) x_j \geq 0$$

Similarly the grade must not exceed 5.0, so we have the further constraint:

$$\sum_{j=1}^{2} (5.0 - GRADE_j) x_j \geq 0$$

Finally there is a limit to the availability of each of the ores. We model this with the constraints:

$$x_1 \leq AVAIL_1$$
$$x_2 \leq AVAIL_2$$

A3.3.2 Developing the Model

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 and the upper/lower limits on the grade of the final product are fixed.

Enter the following model into a file `blend2.mos`.

```
model Blend2
uses "mmxprs"

declarations
 ROres = 1..2           ! Range of Ores
 REV = 125.0            ! Unit revenue of product
 MINGRADE = 4.0         ! Min permitted grade of product
 MAXGRADE = 5.0         ! Max permitted grade of product
 COST: array(ROres) of real    ! Unit cost of ores
 AVAIL: array(ROres) of real   ! Availability of ores
 GRADE: array(ROres) of real
   ! Grade of ores (measured per unit of mass)

 x: array(ROres) of mpvar      ! Quantities of ores used
end-declarations

! Read data from file blend.dat in subdirectory Data
initializations from 'Data/blend.dat'
 COST
 AVAIL
 GRADE
end-initializations

! Objective: maximize total profit
 Profit:= sum(o in ROres) (REV-COST(o))* x(o)

! Lower and upper bounds on ore quality
 LoGrade:= sum(o in ROres) (GRADE(o)-MINGRADE)*x(o) >= 0
 UpGrade:= sum(o in ROres) (MAXGRADE-GRADE(o))*x(o) >= 0

! Set upper bounds on variables
 forall(o in ROres) x(o) <= AVAIL(o)

 maximize(Profit)              ! Solve the LP-problem

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

end-model
```

A3.3 Data from Text Files

The file `Data/blend.dat` contains

```
! 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 'Data/blend.dat'
 AVAIL GRADE COST
end-initializations
```

Section 5.4 has more about getting data from text files into multi-dimensional arrays.

The `mmetc` library can be used in Mosel programs to load data files written for Xpress^{MP}'s `mp-model` using the `diskdata()` procedure. For example, if we had a file `Data/cost.dat` containing data

```
85, 93
```

then the Mosel model

```
model Blendx
 uses "mmxprs", "mmetc"
 declarations
  COST: array(1..2) of real
 end-declarations

 diskdata(ETC_DENSE, "Data/cost.dat", COST)
 writeln("COST(1) is ", COST(1), " COST(2) is ", COST(2))
end-model
```

will output

```
COST(1) is 85 COST(2) is 93
```

A3.3.1 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 `Data/ncost.dat`, then we would have to edit the model, and recompile it. To help with this situation, Mosel has parameters. 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 Blendy
 uses "mmxprs", "mmetc"
 parameters
  CostFile="Data/cost.dat"
 end-parameters
 declarations
  COST: array(1..2) of real
 end-declarations
 diskdata(ETC_DENSE, CostFile, COST)
 writeln("COST(1) is ", COST(1), " COST(2) is ", COST(2))
end-model
```

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

```
mosel -c "load blendy; run 'CostFile="data/ncost.dat "'"
```

will read the cost data from the file we want.

Part A - Chapter 4: Using Databases and Spreadsheets

A4.1 Overview

This chapter shows how Mosel's SQL/ODBC facilities can be used to obtain data from and export data to a variety of spreadsheets and databases.

It is quite easy to create and maintain data tables in text files but we have found that a much better data storage medium is provided by spreadsheets or 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. It requires an additional authorization in your Xpress^{MP} license.

A4.2 A Spreadsheet Example

A4.2.1 Spreadsheets are not Databases

SQL and ODBC were designed for databases, not for spreadsheets. One of the many differences between spreadsheets and databases is that columns in a relational database have a type (text, integer, real, etc.), whereas individual cells in a spreadsheet are typed, and all the cells in a column do not necessarily have the same type. If you are constructing spreadsheet ranges to hold your data for use by ODBC, you must ensure that all the cells in a column of the range have the same type.

Furthermore, if you are writing to a spreadsheet you must be aware that ODBC has to guess the type of the column - and hence the type of all the cells in the column - somehow. It does this by looking at the first few rows in the range. So these rows have to be populated with specimen data.

A4.2.2 The Example

Please note that if you are going to work through the examples in this section, you must have access to Excel.

Let us suppose that in a spreadsheet called `myss.xls` you have inserted the following into the cells indicated:

	A	B	C
1			
2		First	Second
3		6.2	1.3
4		-1.0	16.2
5		2.0	-17.9

and called the range `B2:C5 MyRange`.

We will use ODBC to extract these data into a Mosel array `TOM(1..3,1..2)`

- In Windows set up a User Data Source called `MSExcel`, by clicking `Add`, selecting `Microsoft Excel Driver (*.xls)`, and filling in the ODBC Microsoft Excel Setup dialog. Click `Options >>` and clear the `Read Only` check box.

The following model will set up the `TOM` array in Mosel and fill it with the data from the Excel range `MyRange`.

```
model ODBCex
 uses "mmodbc"
 declarations
  TOM: array(1..3,1..2) of real
 end-declarations

 SQLconnect('DSN=MSExcel; DBQ= Data/myss.xls')
 SQLexecute("select * from MyRange ",[TOM])

 forall(i in 1..3) do
  writeln("Row(",i,"): ", TOM(i,1), " ", TOM(i,2))
 end-do
end-model
```

Note that we use the `mmodbc` package. The ODBC statement "`select * from MyRange`" says "select everything from the range called `MyRange`". ODBC uses SQL, and it is possible to have much more complex selection statements than the ones we have used.

A4.3 A Database Example

If we use Microsoft Access, we might have set up an ODBC DSN called `MSAccess`, and suppose we are extracting data from a table called `MyTable` in the database `moseg.mdb`. There are just two double columns in `MyTable`, called `First` and `Second`. We have just three records in `MyTable`, and the data are the same as in the Excel example above.

We modify the example above to be

```
model ODBCexAC
 uses "mmodbc"
 declarations
  TOM: array(1..3,1..2) of real
 end-declarations

 SQLconnect('DSN=MSAccess; DBQ= Data/moseg.mdb')
 SQLexecute("select * from MyTable ",[TOM])

 forall(i in 1..3) do
  writeln("Row(",i,"): ", TOM(i,1), " ", TOM(i,2))
 end-do
end-model
```

and get the output

```
Row(1): 6.2 1.3
Row(2): -1 16.2
Row(3): 2 -17.9
```

A4.4 Using ODBC to Output Data

It is important to recall the warning in section 4.2.1. that " Spreadsheets are not Databases" Remember in particular that if you are writing to a spreadsheet you must be aware that ODBC has to guess the type of the column - and hence the type of all the cells in the column - somehow. It does this by looking at the first few rows in the range. So these rows have to be populated with specimen data.

A4.4.1 A Database Example

We will take the Microsoft Access example of reading data using ODBC we used above, and show how to write data to a table. We are going to square the numbers we get in `TOM` and write the squares to a table called `MyOut` in the database `Data\mosegout.mdb`. There are just two double columns in `MyOut`, called `First2` and `Second2`.

```
model ODBCOut
 uses "mmodbc"
 declarations
  TOM: array(1..3,1..2) of real
 end-declarations

 SQLconnect('DSN=MSAccess; DBQ=Data\mosegout.mdb')
! Get the data from MyTable
 SQLexecute("select * from MyTable ",[TOM])

! Square all the numbers ...
 forall(i in 1..3, j in 1..2) TOM(i,j) := TOM(i,j)^2

! ... and write to MyOut
 SQLexecute("insert into MyOut (First2, Second2) values (?,?)", TOM)

end-model
```

A4.5 Helpful Tip: Sizing TABLES for Spreadsheet Data

There is a trick that we have found to be very useful in practice. Mosel does the evaluation of what it sees at the point that it sees it. This is particularly helpful when you want to make dynamic adjustments to the sizes of tables Below we develop a set of commands that we use repeatedly when we are feeding Mosel with data from a spreadsheet that obtain the sizes of tables directly from the data source.

Suppose that we have set up some data in a spreadsheet `ssxmpl` to represent the resource usage of some raw materials by some products. We want to be able to allow for the number of raw materials and the number of products changing. Diagrammatically, we have decided to lay out this part of the spreadsheet as follows:

	RawMat1	RawMat2	RawMat3	RawMat4
Product1				
Product2				
Product3				
...				

and to call the range, including the row containing raw material names (but not the column containing product names), `USAGE`. If an extra product is introduced, then `USAGE` gets bigger by one row; if an extra raw material is used then the number of columns increases by one. These changes have to be mirrored in the Mosel formulation.

We construct a small region of the spreadsheet and name it `SIZES`. Into it we put the numbers that characterize the problem - in this case the number of products and the number of raw materials - the parameters. It is important that these numbers are not "hard-wired" in, but that we let the spreadsheet calculate them for itself. In Excel it would be:

Number of Products	Number of Raw Materials
=ROWS(USAGE)-1	=COLUMNS(USAGE)

The reduction by 1 allows for the row which just contains the raw material names. We name as `Nprod` the range formed by the two cells

Number of Products
=ROWS(USAGE)-1

and likewise the range made from the two cells that form the second column of `SIZES` is named `Nrm`.

Now the commands below might form the introductory part of a model.

```
model sizes
 uses "mmodbc"
 declarations
  Nprod, Nrm: integer
 end-declarations

 SQLconnect('DSN=MSExcel;DBQ=Data/ssxmpl.xls')
 Nprod:=SQLreadinteger("select Nprod from SIZES")
 Nrm  :=SQLreadinteger("select Nrm   from SIZES")

 declarations
  PneedsR: array(1..Nprod,1..Nrm) of real
 end-declarations

 SQLexecute("select * from USAGE",[PneedsR])

 forall(p in 1..Nprod, r in 1..Nrm) do
  writeln("PneedsR(",p,",",r, " is ", PneedsR(p,r) )
 end-do
end-model
```

Part A - Chapter 5: More Advanced Modeling Features

A5.1 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***.

The main areas not yet covered in any detail are

- conditional generation
- sparse data
- displaying data
- built in functions
- more on reporting results

The following sections deal with each of these in turn.

A5.2 Conditional Generation

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

$$\sum_{\substack{i=1 \\ s.t. A(i)>0}}^{IMAX} x_i \leq 15$$

In other words, we just want to include in a sum those x_i for which A(i) is greater than zero. This is accomplished by

```
CC:= sum((i in 1..IMAX | A(i)>0 ) x(i) <= 15
```

We can conditionally create variables by a slightly more complex procedure. Suppose that we have a set of 15 decision variables `x(i)` where we do not know the set of `i` for which `x(i)` exist until we have read data into an array `EXIST(i)`.

```
model doesx
 declarations
  IR = 1..15
  EXIST: set of integer
  x: dynamic array(IR) of mpvar
 end-declarations

! Read data from file
 initializations from 'Data/idata.dat'
  EXIST
 end-initializations

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

! Build a little model to show what esists
 obj:= sum(i in IR) x(i)
 c:= sum(i in IR) i * x(i) >= 5

 exportprob(0, "", obj) ! show model
end-model
```

If the data in `idata.dat` are

```
EXIST: [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(0, "", obj)` is a nice idiom for seeing on-screen the problem that has been created.

The key point is that `x` has been declared as a *dynamic* array, and then the ones that exists 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.

Another way to do this, is

```
model doesx2
 declarations
  EXIST: set of integer
 end-declarations

 initializations from 'Data/idata.dat'
  EXIST
 end-initializations
 finalize(EXIST)

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

 obj:= sum(i in EXIST) x(i)
 forall(i in EXIST)  x(i) <= i

 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*. 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 doesn't require us to think of the upper limit of the range `IR` *a priori.*

A5.3 Initializing Multi-Dimensional Arrays

How does one initialize a 2-dimensional array such as

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

For a 1-dimensional array we might have

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

but for our array EE 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. The model

```
model sp
 declarations
  TT: array(1..2, 1..3, 1..4) of integer
 end-declarations

 TT:= [111, 112, 113, 114,
       121, 122, 123, 124,
       131, 132, 133, 134,
       211, 212, 213, 214,
       221, 222, 223, 224,
       231, 232, 233, 234]

 forall(i in 1..2, j in 1..3, k in 1..4) do
  writeln("TT(", i, ",",j, ",", k, ") = ", TT(i,j,k) )
 end-do
end-model
```

produces output

```
TT(1,1,1) = 111
TT(1,1,2) = 112
TT(1,1,3) = 113
TT(1,1,4) = 114
TT(1,2,1) = 121
TT(1,2,2) = 122
TT(1,2,3) = 123
TT(1,2,4) = 124
TT(1,3,1) = 131
TT(1,3,2) = 132
TT(1,3,3) = 133
TT(1,3,4) = 134
TT(2,1,1) = 211
TT(2,1,2) = 212
TT(2,1,3) = 213
TT(2,1,4) = 214
TT(2,2,1) = 221
TT(2,2,2) = 222
TT(2,2,3) = 223
TT(2,2,4) = 224
TT(2,3,1) = 231
TT(2,3,2) = 232
TT(2,3,3) = 233
TT(2,3,4) = 234
```

A5.4 Dynamic Arrays

A5.4.1 Sparsity

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.

```
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: array(PLANT,REGION) of real
   ! Capacity on each route plant->region
  DISTANCE: array(PLANT,REGION) of real
   ! Distance of each route plant->region
  FUELCOST: real                     ! Fuel cost per unit distance
  flow: array(PLANT,REGION) of mpvar ! Flow on each route
 end-declarations

 ! Read data from file
 initializations from 'Data/transprt.dat'
  DEMAND
  PLANTCAP
  PLANTCOST
  DISTANCE
  TRANSCAP
  FUELCOST
 end-initializations

 ! Create the flow variables that exist
 forall(p in PLANT, r in REGION | TRANSCAP(p,r) >0 ) create(flow(p,r))

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

 ! Limits on plant capacity
  forall(p in PLANT) Supply(p):= sum(r in REGION) flow(p,r) <=
    PLANTCAP(p)
 ! Satisfy all demands
  forall(r in REGION) Demand(r):= sum(p in PLANT) flow(p,r) = DEMAND(r)

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

  minimize(MinCost)                  ! Solve the LP-problem

 end-model
```

`REGION` and `PLANT` are declared to be sets of strings, as yet of unknown size. `DEMAND`, `PLANTCAP`, `PLANTCOST` and `TRANSCAP` are arrays that will be indexed by members of `REGION` and `PLANT` The data file `Data/transprt.dat` contains the problem specific data. It might have, for instance,

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

PLANTCAP: [ (Corby) 3000 (Deeside) 2700 (Glasgow) 4500 (Oxford) 4000 ]

PLANTCOST: [ (Corby) 1700 (Deeside) 1600 (Glasgow) 2000 (Oxford) 2100 ]

DISTANCE: [
(Corby   North)     400
(Corby   SWest)     400
(Corby   SEast)     300
(Corby   Midlands) 100
(Deeside Scotland) 500
(Deeside North)   200
(Deeside SWest)   200
(Deeside SEast)   200
(Deeside Midlands) 400
(Glasgow Scotland) 200
(Glasgow North)   400
(Glasgow SWest)   500
(Glasgow SEast)   900
(Oxford  Scotland) 800
(Oxford  North)   600
(Oxford  SWest)   300
(Oxford  SEast)   200
(Oxford  Midlands) 400
]

TRANSCAP: [
(Corby   North)   1000
(Corby   SWest)   1000
(Corby   SEast)   1000
(Corby   Midlands) 2000
(Deeside Scotland) 1000
(Deeside North)   2000
(Deeside SWest)   1000
(Deeside SEast)   1000
(Deeside Midlands) 300
(Glasgow Scotland) 3000
(Glasgow North)   2000
(Glasgow SWest)   1000
(Glasgow SEast)   200
(Oxford  Scotland) 0
(Oxford  North)   2000
(Oxford  SWest)   2000
(Oxford  SEast)   2000
(Oxford  Midlands) 500
]

FUELCOST: 17
```

Note that some data are not specified; for instance, there is no Corby<->Scotland route. So the data are sparse.

A5.4.2 Reading Sparse Data

Suppose we want to read in data of the form

```
i , j, value_ij
```

from an ASCII file, setting up a dynamic array `A(range, range)` just with A(i,j)= value_ij for the (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 show using `writeln()` that the arrays hold identical data.

```
model trio
uses "mmetc"  ! required for diskdata()

declarations
 A1, A2, A3: array(range,range) of real
 i, j: integer
end-declarations

! First method: use an initializations block
 initializations from "Data/data_1.dat"
  A1 as "mydata"
 end-initializations

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

! Third method: use diskdata
 diskdata(ETC_IN+ETC_SPARSE,"Data/data_3.dat", A3)

! Now let's see what we have
 writeln('A1 is: ', A1)
 writeln('A2 is: ', A2)
 writeln('A3 is: ', A3)

end-model
```

The data files could be set up thus:

data_1.dat

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

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
```

data_3.dat

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

Note that the second way of setting up and accessing data demonstrates the immense flexibility of `readln`.

A5.5 Useful Functions and Procedures

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

Accessing solution values	`getsol, getact, getcoeff, getdual, getrcost, getslack, getobjval`
Arithmetic functions	`arctan, cos, sin, ceil, floor, round, exp, ln, log, sqrt, isodd`
List functions	`maxlist, minlist`
String functions	`strfmt, substr`
Dynamic array handling	`create, finalize`
File handling	`fclose, fflush, fopen, fselect, fskipline, getfid, iseof, read, readln`
Accessing control parameters	`getparam, setparam`
Getting information	`getsize, gettype, getvars`
Hiding constraints	`sethidden, ishidden`
Miscellaneous functions	`exportprob, bittest, random, setcoeff, settype, exit`

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

Optimize	`minimize, maximize`
MIP directives	`setmipdir, clearmipdir`
Handle bases	`savebasis, loadbasis, delbasis`
Force problem loading	`loadprob`
Get 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 XPRS.

```
declarations
  status:array({XO_OPT,XO_UNF,XO_INF,XO_UNB}) of string
end-declarations

status:=["Optimum found","Unfinished","Infeasible","Unbounded"]
...
minimize(obj)
writeln(status(getprobstat))
```

In the `mmsystem` module are various useful functions provided by the underlying operating system:

Delete a file/directory	`fdelete, removedir`
Move a file	`fmove`
Current working directory	`getcwd`
Get an environment variable's value	`getenv`
File status	`getfstat`
Returns the system status.	`getsysstat`
Time	`gettime`
Make a directory	`makedir`
General system call	`system`

See the ***Mosel Reference Manual*** for full details.

Part A - Chapter 6: Integer Programming

A6.1 Introduction to 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 (global) entities* and then the Xpress^{MP} 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.

Xpress^{MP} handles the following global entities:

Binary variables (BV) - decision variables that can take either the value 0 or the value 1 (do/don't do variables).

Integer variables (UI) - decision variables that can take only integer values. Some small upper limit must be specified.

Partial integer variables (PI) - decision variables that can take integer values up to a specified limit and any value above that limit.

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

Semi-continuous integer variables (SI) - decision variables that can take either the value 0, or an integer value between some lower limit and upper limit. SIs 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.

Special Ordered Sets of type one (SOS1 or S1) - an ordered set of variables at most one of which can take a non-zero value.

Special Ordered Sets of type two (SOS2 or S2) - an ordered set of variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

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 aero plane 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 mi-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).

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 global 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, a small example follows. The first formulation uses binary variables. This formulation is then modified use Special Ordered Sets.

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

A6.2 A Project Planning Model

The problem to be modeled is as follows:

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.

We define the following constants:

NPROJ:	the number of projects; and
NTIME:	the number of months to plan for.

The data are:

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

and the variables:

x_{pm}:	=1 if project p starts in month m, otherwise 0
$start_p$:	start month for project p

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 $m+DUR_p-1$. So, in total, we get the benefit of BEN_p for $NMTH-(m+ DUR_p-1) = NMTH - m- DUR_p+1$ 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 $NMTH-DUR_p$. Thus the profit is:

$$profit = \sum_{p=1}^{NPROJ} \sum_{m=1}^{NMTH-DUR_p} BEN_p(NMTH - m - DUR_p + 1)x_{pm}$$

Each project must be done once, so it must start in one of the months 1 to $NMTH\text{-}DUR_P$:

$$\sum_{m=1}^{NMTH-DUR_P} x_{pm} = 1 \ \forall p$$

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 $(k\text{-}m+1)^{th}$ month in month k, and so will be using $PROF_{p,k-m+1}$ 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:

$$\sum_{p=1}^{NPROJ} \sum_{m=1}^{k} PROF_{p,k+1-m} x_{pm} \le RESMAX_k \ \forall k$$

The start month of a project is given by:

$$\sum_{m=1}^{NMTH-DUR_P} m x_{pm} = start_p \ \forall p$$

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

Finally we have to specify that the x_{pm} are binary (0 or 1) variables. This is done by the statement:

$$x_{pm} \in \{0,1\} \ \forall p, m$$

The model as specified to Mosel is as follows:

```
model Pplan
 uses "mmxprs"

 declarations
  RProj = 1..3            ! Range of projects
  NMTH = 6                ! Time horizon (months)
  RMonth = 1..NMTH        ! Range of time periods (months) to plan for

  DUR: array(RProj) of integer    ! Duration of project p
  PROF: array(RProj,RMonth) of integer
             ! Resource usage of project p in its t'th month
  RESMAX: array(RMonth) of integer
             ! Resource available in month t
  BEN: array(RProj) of real  ! Benefit per month once project finished

  x: array(RProj,RMonth) of mpvar
       ! 1 if proj p starts in mnth t, else 0

  start: array(RProj) of mpvar   ! Month in which project p starts
 end-declarations

 DUR   := [3, 3, 4]
 RESMAX:= [5, 6, 5, 5, 4, 5]
 BEN   := [10.2, 12.3, 11.2]
 PROF(1,1):= [3, 4, 2]
 PROF(2,1):= [4, 1, 6]
 PROF(3,1):= [3, 2, 1, 2] ! Other PROF 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 NMTH-(t+DUR(p)-1) months:
 MaxBen:=
  sum(p in RProj, m in 1..NMTH-DUR(p)) (BEN(p)*(NMTH-m-DUR(p)+1)) * x(p,m)
```

```
! Resource availability
! A project starting in month m is in its k-m+1'st month in month k:
 forall(k in RMonth) ResMax(k):=
  sum(p in RProj, m in 1..k) PROF(p,k+1-m)*x(p,m) <= RESMAX(k)

! Each project starts once and only once:
 forall(p in RProj) One(p):= sum(m in RMonth) x(p,m) = 1.0

! Connect variables x(p,t) and start(p)
 forall(p in RProj) Connect(p):= sum(m in 1..NMTH-DUR(p)) m*x(p,m) = start(p)

! Make all the x variables binary
 forall(p in RProj, m in RMonth) x(p,m) is_binary

 maximize(MaxBen)                ! Solve the MIP-problem
 writeln("Solution value is: ", getobjval)

end-model
```

A6.3 The Project Planning Model Using Special Ordered Sets

The example can be modified to use Special Ordered Sets of type 1 (SOS1). The x_{mt} 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 `x(p,m)` in the specification of the SOS. For example, x_{p1}'s coefficient, 1, is less than x_{p2}'s, 2, which in turn is less than x_{p3}'s coefficient, and so on

The fact that the x_{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 x is non-zero for
   each project p. Use month index to order the variables !)

forall(p in RProj) XSet(p):= sum(m in RTime) m*x(p,m) is_sos1
```

The `is_sos1` specification tells Mosel that `Xset(p)` is a Special Ordered Set of type 1.The `sum` term is not a summation operator but really Mosel's equivalent to the set theoretic union concept. In this context it says that all the `x(p,m)` variables for m in the `RTime` index range are members of an SOS1 with reference row entries t. If the set were an SOS2 set then the `is_sos1` specification would be replaced by `is_sos2`.

The specification of the `x(p,m)` as binary variables must now be removed. The binary nature of the `x(p,m)` is implied by the SOS1 property, since if the `x(p,m)` 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 Program (LP) that is solved as a first approximation to an Integer Program gives an answer where x_{p1} is fractional, say with a value of 0.5, and x_{pM} 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 x_{p1} or $x_{pT.}$ 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 x_{pm} are ordered and the LP 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 x_{pm} to 0, and restricting the choice of the one x_{pm} that can be 1 to the later x_{pm}. It has this option because it now has the information to "know" what is an early and what is a late x_{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^{MP} it is often found to be an order of magnitude better than the equivalent binary formulation for large problems.

Part B

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 been briefly mentioned in Part A, are covered in a more detailed way:

selections and loops

working with sets

output to files and producing formatted output

functions and procedures

Whilst the first 4 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.

Part B - Chapter 1: 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 take a closer look at two fundamental types of control constructs in Mosel: selections and loops.

Actions in a program frequently 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.

B1.1 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-then` statement:

B1.1.1 `if-then`

"If a condition holds, do something". For example:

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

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

Note that there may be any number of expressions between `then` and `end-if`, not just a single one. In other cases, it may be necessary to express choices with alternatives.

B1.1.2 `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 than or equal to 20, otherwise the lower bound 35 is applied to it.

B1.1.3 `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 than or equal to 20, and if the value of `A` is less than or equal to 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.

B1.1.4 `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-if
```

Here still the upper bound 7 is applied to the variable `x` if the value of `A` is greater than or equal to 20, and the lower bound 35 is applied if the value of `A` is less than or equal to 10. In addition, `x` is fixed to 1 if `A` has a value 12 or 15, and fixed to 0 for all remaining values.

An example of the use of the `case` statement is given in Chapter B-5.

The following example 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).

B1.2 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`, `while`, `while-do`, and `repeat-until`.

B1.2.1 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 an index of a forall loop.

The following example 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
       sumd+=d
     end-if
   if p=sumd then
     writeln(p)
   end-if
 end-do

end-model
```

The outer loop encloses several statements, so we 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.

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
```

Conditional Looping

The possibility of adding conditions to a forall loop via the `|" symbol has already been mentioned in Chapter A-5. Conditions may be applied to one or several indices and the selection statement(s) can be placed accordingly.

Consider the following example where A and U are one- and two-dimensional arrays of integers and reals respectively, and y is 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)
```

B1.2.1 `while`

A `while` loop is typically employed if the number of times that the loop needs to be executed is not known 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 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
```

B1.2.3 `repeat until`

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

The following example combines the three types of loops (`forall`, `while`, `repeat-until`) that are available in Mosel. It implements a Shellsort 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 *i(1)=1, i(k+1)=3i(k)+1, k=1,2,...*

```
model ShellSort
 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 then break; end-if
     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 just as well write `break 1`. If we wrote `break 3`, the break would make the algorithm jump 3 levels of looping higher, that is outside of the `repeat-until` loop.

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

Part B - Chapter 2: Sets

A set collects objects of the same type without establishing an order among them (which is the case with arrays). In Mosel, sets may be defined for all elementary types, that is the basic types (`integer`, `real`, `string`, `boolean`) and the MP types (`mpvar` and `linctr`).

This chapter presents in a more systematic way the different ways sets may be initialized (all of which the reader has already encountered in the examples in Part A), and also shows more advanced ways of working with sets.

B2.1 Initializing Sets

In the revised formulation of the Burglar Problem in Chapter 3 and in the models in Chapter A-5 we have already seen different examples of the use of index sets. We recall here the relevant parts of the respective models.

B2.1.1 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 are 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 need to be assigned after its declaration:

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

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

B2.1.2 Set Initialization from File, Finalized and Fixed Sets

In Chapter 5 the reader encountered several examples of 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
 EXIST: set of integer
end-declarations

initilizations from 'Idata.dat'
 EXIST
end-initializations
```

where `Idata.dat` contains data in the following format:

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

Unless a set is constant, arrays that are indexed by this set are created as dynamic arrays. Since in many cases the contents of a set do not change further after its initialization, Mosel provides the `finalize` statement that turns a (dynamic) set into a constant set. Consider the continuation of the example above:

```
finalize(EXIST)

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

The array of variables `x` will be created as a static array; without the `finalize` statement it would be dynamic since the index set `EXIST` may still be subject to changes. Declaring arrays in the form of static arrays is preferable if the indexing set is known beforehand because this allows Mosel to handle them more efficiently.

Index sets may also be initialized indirectly during the initialization of dynamic 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 where data is input from file. In the following example we add an array of variable descriptions to the chess problem introduced in the first chapter of this manual. These descriptions may, for instance, be used for generating nice output. Since the array `DescrV` and its indexing set `Allvars` are dynamic they grow with each new variable description that is added to `DescrV`.

```
model Chess3
 uses "mmxprs"

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

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

 profit:=  5*small + 20*large
 mc_time:= 3*small + 2*large <= 400
 wood:=      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 noted 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`.

B2.2 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 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 union of all 3 sets
 In_all_three:= Cities*Ports*Capitals ! and 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',`liverp
ool',`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 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.

B2.3 Set Operators

The preceding example introduced 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 (`>=`), different (`<>`), end equality (`=`). Their use is illustrated by the following example:

```
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
```

Part B - Chapter 3: 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, whereas *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 Reference Manual), and it is possible to define new functions and procedures according to the needs in a specific program. A procedure that has occurred repeatedly in this document is `writeln`. Typical examples of functions are mathematical functions like `abs`, `floor`, `ln`, `sin` etc.

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 "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
```

B3.1 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 parentheses following 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* local 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".

Whilst it is not possible to modify function/procedure parameters in the corresponding subroutine, the declaration of local parameters may *hide* these parameters. Mosel considers this as a likely mistake and prints a warning during compilation (without any consequence for the execution of the program).

```
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 warning

```
Mosel: W-165 at (30,3) of `subrout.mos': Declaration of `a' hides a parameter.
```

which is due to the redefinition of the parameter in procedure `hide_a_3`. The program 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
```

B3.2 Recursion

following example 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 below).

B3.3 `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`. The *declaration* 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 implements a quick sort algorithm for sorting a randomly generated array of numbers into ascending order. The procedure `qsort` that starts the sorting algorithm is defined at the very end of the program, so it needs to be declared at the beginning, before it is called. Procedure `qsort_start` 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 another subroutine, `swap`.

The idea of the quicksort 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 qsort_start(L:array(range) of integer)

 declarations
  T:array(1..LIM) of integer
 end-declarations

 forall(i in 1..LIM) T(i):=round(.5+random*LIM)
 writeln(T)
 qsort_start(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 then qsort(L,s,j); end-if
  if i>s and i<e then qsort(L,i,e); end-if
 end-procedure

! Start of the sorting process
 procedure qsort_start(L:array(r:range) of integer)
  qsort(L,getfirst(r),getlast(r))
 end-procedure

end-model
```

The quicksort example above demonstrates typical uses of subroutines, namely regrouping 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 `qsort_start`): in Mosel there is no limit on the number of nested calls to subroutines (it is not possible, though, to define subroutines within a subroutine).

B3.4 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 "Quick Sort 2"

 parameters
  LIM=50
 end-parameters

 forward procedure qsort(L:array(range) of integer)

 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 `qsort_start` 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. On 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 no longer be possible to differentiate between these two procedures.

Part B - Chapter 4: Output

B4.1 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 more easily readable output, these procedures can be combined with the formatting procedure `strfmt` that indicates the (minimum) space reserved for printing an item and its placement within this space (negative values mean left justified printing, positive right justified).

The following example prints out the solution of model Transport (Chapter A-5) 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 printtab
 declarations
  rsum: array(REGION) of integer    ! Data table for printing
  psum,prsum,ct,iflow: integer      ! Counters
 end-declarations

        ! Print heading and the first line of the table
 writeln("\nProduct Distribution\n--------------------")
 writeln(strfmt("Sales Region",44))
 write(strfmt("",14))
 forall(r in REGION) write(strfmt(r,9))
 writeln(strfmt("TOTAL",9), " Capacity")
        ! Print the solution values of the flow variables and
        ! calculate totals per region and per plant
 ct:=0
 forall(p in PLANT) do
   ct += 1
   if ct=2 then
     write("Plant ",strfmt(p,-8))
   else
     write("      ",strfmt(p,-8))
   end-if
   psum:=0
   forall(r in REGION) do
     iflow:=integer(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,9), strfmt(integer(PLANTCAP(p)),9))
 end-do

        ! Print the column totals
 write("\n", strfmt("TOTAL",-14))
 prsum:=0
 forall(r in REGION) do
   prsum += rsum(r);
   write(strfmt(rsum(r),9))
 end-do
 writeln(strfmt(prsum,9))
```

```
          ! Print demand of every region
    write(strfmt("Demand",-14))
    forall(r in REGION) write(strfmt(integer(DEMAND(r)),9))

          ! Print objective function value
    writeln("\n\nTotal cost of distribution = ",strfmt(getobjval/1e6,0,3)," million.")
   end-procedure
```

With the data from Chapter A-5 the procedure produces the following output:

```
Product Distribution
--------------------
                              Sales Region
               Scotland    North    SWest    SEast Midlands    TOTAL Capacity
      Corby                            180      820     2000     3000     3000
Plant Deeside                 1530      920              250     2700     2700
      Glasgow      2840     1270                               4110     4500
      Oxford                         1500     2000      500     4000     4000

TOTAL              2840     2800     2600     2820     2750    13810
Demand             2840     2800     2600     2820     2750

Total cost of distribution = 81.018 million.
```

B4.2 File Output

If we do not want the output of procedure `printtab` in the previous section to be displayed on screen but to be saved in the file `solout.txt`, we simply open the file for writing at the beginning of the procedure by adding

```
fopen("solout.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 `solout.txt` to be deleted, so that the table is appended at the end of the file, we need to write the following for opening the file (closing it the same way as before):

```
fopen("solout.txt",F_OUTPUT+F_APPEND)
```

Similarly to the input of data from file, in Mosel there are several ways of outputting data (e.g. solution values) to a file. The following example demonstrates three different ways of writing the contents of a table `A` to a file.

```
model "trio output"
 uses "mmetc"

 declarations
  A: array(1..3,1..3) 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

! Second method: use the built-in writeln function
 fopen("out_2.dat",F_OUTPUT)
 forall(i,j in 1..3)
  writeln('A_out(',i,' and ',j,') = ', A(i,j))
 fclose(F_OUTPUT)

! Third method: use diskdata
 diskdata(ETC_OUT+ETC_SPARSE,"out_3.dat", A)

end-model
```

File `out_1.dat` will contain the following:

```
'myout': [2 4 6 12 14 16 22 24 26]
```

If this file already contains 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.

The nicely formatted output to `out_2.dat` results in the following:

```
A_out(1 and 1) = 2
A_out(1 and 2) = 4
A_out(1 and 3) = 6
A_out(2 and 1) = 12
A_out(2 and 2) = 14
A_out(2 and 3) = 16
A_out(3 and 1) = 22
A_out(3 and 2) = 24
A_out(3 and 3) = 26
```

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:

```
1,1,2
1,2,4
1,3,6
2,1,12
2,2,14
2,3,16
3,1,22
3,2,24
3,3,26
```

Without option `ETC_SPARSE` `out_3.dat` looks as follows:

```
2,4,6
12,14,16
22,24,26
```

Part B - Chapter 5: 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.

B5.1 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.

B5.1.1 Example Problem

The problem we want to solve is the following: a large company is planning to outsource the cleaning of its offices at least cost. The *NSITES* office sites of the company are grouped into *NAREAS* areas. Several professional cleaning companies (with a total number of *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 dependency on a single contractor, adjacent areas have to be allocated to different contractors. Every site s (s in 1,...,*NSITES*) is to be allocated to a single contractor, but there may be between $LOWCON_a$ and $UPPCON_a$ contractors per area a.

B5.1.1.1 Problem Formulation

The problem stated above may be summarized as follows:

Objective:

Minimize the total cost of all contracts.

Constraints:

Each site must be cleaned by exactly one contractor.

Adjacent areas must not be allocated to the same contractor.

The lower and upper limits on the number of contractors per area must be respected.

For the mathematical formulation of the problem we introduce two sets of variables:

x_{cs} indicates whether contractor c is cleaning site s

y_{ca} indicates whether contractor c is allocated any site in area a

To express the relation between these two sets of variables we need another collection of 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.

```
model "Office cleaning"

 uses "mmxprs","mmsystem"

 forward procedure cutgen

 declarations
  PARAM: array(1..3) of integer
 end-declarations

 initializations from 'Data/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
  RA = 1..NAREAS
  RC = 1..NCONTRACTORS
  RS = 1..NSITES
  AREA: array(RS) of integer        ! Area site is in
  NUMSITE: array(RA) of integer     ! Number of sites in an area
  LOWCON: array(RA) of integer      ! Lower limit on the number of
                                    ! contractors per area
  UPPCON: array(RA) of integer      ! Upper limit on the number of
                                    ! contractors per area
  ADJACENT: array(RA,RA) of integer ! =1 if areas adjacent
  PRICE: array(RS,RC) of real       ! Price per contractor per site

  x: dynamic array(RC,RS) of mpvar  ! 1 iff c allocated to site s
  y: array(RC,RA) of mpvar          ! 1 iff ctrctor allocated to
                                    !  a site in area a
 end-declarations

 initializations from 'Data/cldata.dat'
  NUMSITE
  LOWCON
  UPPCON
  ADJACENT
  PRICE
 end-initializations

 ct:=1
 forall(a in RA) do
  forall(s in ct..ct+NUMSITE(a)-1)
   AREA(s):=a
  ct+= NUMSITE(a)
 end-do

 forall(c in RC)
  forall(s in RS | PRICE(s,c) > 0.01)
   create(x(c,s))

      ! Objective: Minimize total cost of all cleaning contracts
  cost:= sum(c in RC, s in RS) PRICE(s,c)*x(c,s)

      ! Each site must be cleaned by exactly one contractor
 forall(s in RS)
  clean(s):= sum(c in RC) x(c,s) = 1

      ! Ban same contractor from serving adjacent areas
 forall(c in RC, a in RA, a2 in RA | a > a2 and ADJACENT(a,a2) = 1)
  adj(c,a,a2):= y(c,a) + y(c,a2) <= 1

      ! Specify lower & upper limits on contracts per area
 forall(a in RA | LOWCON(a)>0)
  area_low(a):= sum(c in RC) y(c,a) >= LOWCON(a)
 forall(a in RA | UPPCON(a)<NCONTRACTORS)
  area_upp(a):= sum(c in RC) y(c,a) <= UPPCON(a)
```

```
     ! Define y[c,a] to be 1 iff some x[c,s]=1 for sites s in area a
  forall(c in RC, a in RA) do
   y_upp(c,a):= y(c,a) <= sum(s in RS| AREA(s)=a) x(c,s)
   y_low_area(c,a):= y(c,a) >= 1.0/NMSITE(a) * sum(s in RS| AREA(s)=a) x(c,s)
  end-do

  forall(c in RC) do
   forall(s in RS) x(c,s) is_binary
   forall(a in RA) y(c,a) is_binary
  end-do

  starttime:= gettime

  cutgen

  minimize(cost);                ! Solve the MIP problem
  writeln("(", gettime-starttime, " sec) Global status ",
     getparam("XPRS_MIPSTATUS"), ", best solution:", getobjval);
 ....
```

In the preceding model, we have chosen to implement the constraints that force the variables y_{ca} to become 1 whenever an x_{cs} is 1 for some site s in area a in an aggregated way (this type of constraint is usually referred to as a Multiple Variable Lower Bound, MVLB, constraint). Instead of

```
forall(c in RC, a in RA)
 y_low_area(c,a):= y(c,a) >= 1.0/NMSITE(a) * sum(s in RS| AREA(s)=a) x(c,s)
```

we could have used the stronger formulation

```
forall(c in RC, s in RS) y_low_site(c,s):= y(c,AREA(s)) >= x(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. 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 `y_low_site` that are violated by the current LP solution.

The cut generation loop (procedure `cutgen`) performs the following steps:

1. solve the LP and save the basis
2. get the solution values
3. identify violated constraints and add them to the problem
4. load the modified problem and load the previous basis

```
procedure cutgen

 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
  ztolrhs:real                ! Zero tolerance
  solx: array(RC,RS) of real  ! Sol. values for variables x
  soly: array(RC,RA) of real  ! Sol. values for variables y
  objval,starttime: real
  cut: array(range) of linctr
 end-declarations

 starttime:=gettime
 setparam("XPRS_CUTSTRATEGY", 0)      ! Disable automatic cuts
 setparam("XPRS_PRESOLVE", 0)         ! Switch presolve off
 ztolrhs:= getparam("XPRS_FEASTOL")   ! Get the zero tolerance
 ztolrhs:= ztolrhs * 10
 ncut:=0
 npass:=0
```

```
  while (ncut<MAXCUTS and npass<MAXPASS) do
    npass+=1
    npcut:= 0
    minimize(XPRS_LIN, cost)           ! Solve the LP
    savebasis(1)                       ! Save the current basis
    objval:= getobjval                 ! Get the objective value

    forall(c in RC) do                 ! Get the solution values
      forall(a in RA) soly(c,a):=getsol(y(c,a))
      forall(s in RS) solx(c,s):=getsol(x(c,s))
    end-do

      ! Search for violated constraints and add them to the problem:
    forall(s in RS)
     forall(c in RC)
      if(solx(c,s)-soly(c,AREA(s)) > ztolrhs) then
       cut(ncut):= y(c,AREA(s)) >= x(c,s)
       ncut+=1
       npcut+=1
       if(npcut>MAXPCUTS or ncut>MAXCUTS) then break 2; end-if
      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(1)                     ! 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("max. num. of passes reached")
  else writeln("no more violations")
  end-if

 end-procedure

end-model
```

B5.2 Column Generation

The technique of column generation is used for solving linear problems with a huge number of variables when it is not practical to generate all columns of the problem matrix explicitly. 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 subsequently is solved by solving a sequence of LPs without starting a tree search.

B5.2.1 Example Problem

A paper mill produces rolls of paper of a fixed width *MAXWIDTH* that are subsequently cut into smaller rolls according to the orders by the customers. The rolls can be cut into *NWIDTHS* different sizes. The orders are given as demand per 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.

B5.2.1.1 Problem 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 ($PATTERN_1$,...,$PATTERN_{NWIDTH}$), that consists of cutting small rolls all of the same width as many times as possible out of the large roll.

When we define variables pat_j to denote the number of time a cutting pattern j is used, the objective is to minimize the sum of these variables, subject to the constraints that the demand for all sizes have to be met.

```
model Papermill

 uses "mmxprs"

 forward procedure colgen
 forward function knapsack(c:array(range) of real,
                           a:array(range) of real,
                           b:real,
                           xbest:array(range) of integer):real
 forward procedure shownewpat(dj:real, vx: array(range) of integer)

 declarations
  NWIDTHS = 5                           ! Number of different widths
  RW = 1..NWIDTHS                       ! Range of widths
  RP: range                             ! Range of cut ting patterns
  MAXWIDTH = 94                         ! Maximum roll width
  EPS = 1e-6                            ! Zero tolerance

  WIDTH: array(RW) of real              ! Possible widths
  DEMAND: array(RW) of integer          ! Demand per width
  PATTERNS: array(RW, RW) of integer    ! (Basic) cutting patterns

  pat: array(RP) of mpvar               ! Rolls per pattern
  solpat: array(RP) of real             ! Solution values for vars
  dem: array(RW) of linctr              ! Demand constraints
  minRolls: linctr                      ! Objective function
  knap_ctr, knap_obj: linctr            ! Knapsack constraint+objective
  x: array(RW) of mpvar                 ! Knapsack variables
 end-declarations

 WIDTH:=  [17, 21, 22.5, 24, 29.5]
 DEMAND:= [150, 96,   48, 108,  227]
                                       ! Make basic patterns
 forall(j in RW) PATTERNS(j,j):= floor(MAXWIDTH/WIDTH(j))

 forall(j in RW) do
  create(pat(j))                       ! Create NWIDTHS variables pat
  pat(j) is_integer                    ! Variables are integer and bounded
  pat(j) <= integer(ceil(DEMAND(j)/PATTERNS(j,j)))
 end-do
 forall(j in RW) x(j) is_integer       ! Knapsack variables are integer
 minRolls:= sum(j in RW) pat(j)        ! Objective function

                                       ! Satisfy all demands
 forall(i in RW) dem(i):= sum(j in RW) PATTERNS(i,j) * pat(j) >= DEMAND(i)

 colgen                                ! Column generation at top node

 minimize(minRolls)                    ! Compute the best integer solution for
                                       ! the current problem (including the
                                       ! new columns)
 writeln("Optimal solution: ", getobjval, " rolls)
 write("   Rolls per pattern: ")
 forall(i in RP) write(getsol(pat(i)),", ")
 writeln
```

With the basic set of cutting patterns the mill can satisfy the demand, 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 function `colgen` 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:

1. solve the LP and save the basis
2. get the solution values
3. compute a more profitable cutting pattern based on the current solution
4. generate a new column (=cutting pattern): add a term to the objective function and to the corresponding demand constraints
5. load the modified problem and load the saved basis

To be able to increase the number of variables pat_j in this function, these variables have been declared at the beginning of the program as a dynamic array without specifying any index range.

```
procedure colgen

 declarations
  dualdem: array(RW) of real
  xbest: array(RW) of integer
  dw, zbest, objval: real
 end-declarations

  setparam("XPRS_CUTSTRATEGY", 0)       ! Disable automatic cuts
  npatt:=NWIDTHS
  npass:=1

  while(true) do
    minimize(XPRS_LIN, minRolls)        ! Solve the LP
    savebasis(1)                        ! Save the current basis
    objval:= getobjval                  ! Get the objective value
                                        ! Get the solution values
    forall(j in 1..npatt)  solpat(j):=getsol(pat(j))
    forall(i in RW)  dualdem(i):=getdual(dem(i))

    zbest:=knapsack(dualdem, WIDTH, MAXWIDTH, xbest) - 1

    write("Pass ", npass, ": ")
    if(zbest < EPS) then
      writeln("no profitable column found.\n")
      break
    else
      shownewpat(zbest, xbest)          ! Print the new pattern
      npatt+=1
      create(pat(npatt))                ! Create a new variable
      pat(npatt) is_integer

      minRolls+= pat(npatt)             ! Add new variable to the objective
      dw:=0
      forall(i in RW)                   ! Add new variable to demand constraints
        if(xbest(i) > EPS) then
         dem(i)+= xbest(i)*pat(npatt)
         dw:= maxlist(dw, ceil(DEMAND(i)/xbest(i) ))
        end-if
      pat(npatt) <= dw                  ! Set upper bound on the new variable

      loadprob(minRolls)                ! Reload the problem
      loadbasis(1)                      ! Load the saved basis
    end-if
    npass+=1
 end-do

end-procedure
```

The preceding procedure `colgen` calls the following auxiliary function `knapsack` to solve an integer knapsack problem of the form

z = max{cx : ax<=b, x is integer}

```
function knapsack(c:array(range) of real,
                  a:array(range) of real,
                  b:real,
                  xbest:array(range) of integer):real

     forall(j in RW) sethidden(dem(j) , true)  ! Hide the demand constraints

     knap_ctr:= sum(j in RW) a(j)*x(j) <= b   ! Define the knapsack constraint
     knap_obj:= sum(j in RW) c(j)*x(j)        ! Define the objective function

     maximize(knap_obj)                       ! Solve the knapsack problem
     returned:=getobjval                      ! Get the objective value
                                              ! Get the solution values
     forall(j in RW) xbest(j):=round(getsol(x(j)))

     knap_ctr:= 0                             ! Delete (reset) knapsack constraint
     knap_obj:= 0                             ! Delete (reset) knapsack objective
     forall(j in RW) sethidden(dem(j), false) ! Unhide demand constraints
 end-function
```

The knapsack problem is a second, completely independent optimization problem that is stated within the same model as the main problem. The formulation above uses a procedure that has not yet been introduced:

```
sethidden(c:linctr, b:boolean)
```

With this procedure, constraints can be removed (`hidden') from the problem that is solved by the optimizer without deleting them in the problem definition. This means that the optimizer solves a subproblem of the complete problem stated in Mosel. In the current case, when solving the knapsack problem the optimizer only takes into account the knapsack constraint. At the end of function `knapsack` the definition of the knapsack problem is removed and the demand constraints are re-activated so that the next call to the optimizer in the loop of function `colgen` will solve the main, cutting stock problem.

The constraints of the knapsack problem are defined globally because problem definition in Mosel is incremental. The constraints defined in a subroutine are not deleted at the end of it (hence the need to reset the knapsack constraint explicitly before leaving function knapsack).

The following procedure is called from procedure `colgen` to print out every new pattern that is found:

```
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 1..NWIDTHS) do
    write(WIDTH(i), ":", vx(i), "  ")
    dw += WIDTH(i)*vx(i)
  end-do
  writeln("Total width: ", dw)
end-procedure
```

Part B - Chapter 6: 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.

B6.1 Recursion

Recursion, more properly known as Successive Linear Programming, 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.

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:

*interest(t) = (92/365) * balance(t) * interest_rate*

The balance at time *t (t=1,...,T)* results from the balance of the previous period *t-1* and the net of payments and interest:

net(t) = Payments(t) - interest(t)

balance(t) = balance(t-1) - net(t)

This problem cannot be modeled just by LP because we have the T products

*balance(t) * 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

*i(t) = 92/365 * (b(t-1) * x)*
*= 92/365 * (b(t-1) * (X + dx))*
*= 92/365 * (b(t-1) * X + b(t-1) * dx)*

We now also replace the balance *b(t-1)* in the product with dx by a guess *B(t-1)* and a deviation *db(t-1)*

*i(t) = 92/365 * (b(t-1) * X + (B(t-1)+db(t-1)) * dx)*
*= 92/365 * (b(t-1) * X + B(t-1) * dx + db(t-1) * dx)*

which can be approximated by dropping the product of the deviation variables

*i(t) = 92/365 * (b(t-1) * X + B(t-1) * dx)*

To ensure feasibility we add penalty variables *epl* and *emn* for positive and negative deviations in the formulation of the constraint:

*i(t) = 92/365 * (b(t-1) * X + B(t-1) * dx + epl - emn)*

The model then looks as follows (note the balance variables `b(t)` as well as the deviation `dx` and the quarterly nets `n(t)` are defined as free variables, that is they may take any values between minus and plus infinity using `is_free`):

```
model Fin_nlp

 uses "mmxprs"

 forward procedure solverec

 declarations
  T=6                                ! Time horizon
  RT=1..T                            ! Range of time periods
  P,R,V: array(RT) of real           ! Payments
  B: array(RT) of real               ! An INITIAL GUESS as to balances b(t)
  X: real                            ! An INITIAL GUESS as to interest rate x

  i: array(RT) of mpvar              ! Interest
  n: array(RT) of mpvar              ! Net
  b: array(RT) of mpvar              ! Balance
  x: mpvar                           ! Interest rate
  dx: mpvar                          ! Change to x
  epl, emn: array(RT) of mpvar       ! + and - deviations
 end-declarations

 X:= 0.0
 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 RT) net(t):= n(t) = (P(t)+R(t)+V(t)) - i(t)

                                    ! Money balance across periods
 forall(t in RT) bal(t):= b(t) = if(t>1, b(t-1), 0) - n(t)

                                    ! Approximation of interest
 forall(t in 2..T)
  interest(t):= -(365/92)*i(t) + X*b(t-1) + B(t-1)*dx + epl(t) - emn(t) = 0

 def:= X + dx = x                   ! Define the interest rate: x = X + dx

 feas:= sum(t in RT) (epl(t)+emn(t)) ! Objective: get feasible

 i(1) = 0                           ! Initial interest is zero
 forall (t in RT) n(t) is_free
 forall (t in 1..T-1) b(t) is_free
 b(T) = 0                           ! Final balance is zero
 dx is_free

 minimize(feas)                     ! Solve the LP-problem

 solverec                           ! Recursion loop

                                    ! Print the solution
 writeln("\nThe interest rate is ", getsol(x))
 write(strfmt("t",5), strfmt(" ",4))
 forall(t in RT) write(strfmt(t,5), strfmt(" ",3))
 write("\nBalances ")
 forall(t in RT)  write(strfmt(getsol(b(t)),8,2))
 write("\nInterest ")
 forall(t in RT)  write(strfmt(getsol(i(t)),8,2))
 writeln

end-model
```

In the above model we have declared the procedure `solverec` that executes the recursion but it has not yet been defined. The recursion on `x` and the `b(t)` (t=1,...,T-1) is implemented by the following steps:

1. The `B(t-1)` in constraints `interest(t)` get the prior value of `b(t-1)`
2. The `X` in constraints `interest(t)` get the prior value of `x`
3. The `X` in constraint `def` gets the prior value of `x`

We say we have converged when the change in `dx` (`variation`) is less than 0.000001 (`TOLERANCE`).

```
procedure solverec

  declarations
   TOLERANCE=0.000001                  ! Convergence tolerance
   variation: real                     ! Variation of x
   BC: array(RT) of real
  end-declarations

  variation:=1.0
  ct:=0

  while(variation>TOLERANCE) do
   savebasis(1)                       ! Save the current basis
   ct+=1
   forall(t in 2..T)
     BC(t-1):= getsol(b(t-1))         ! Get solution values for b(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)*b(t-1)
   end-do
   def+= XC-X
   X:=XC
   oldxval:=XC                        ! Store solution value of x

   loadprob(feas)                     ! Reload the problem into optimizer
   loadbasis(1)                       ! 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).

B6.2 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 Archimedean 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 Archimedean model weights or penalties for not achieving targets must be specified, and we attempt to minimize the sum of the weights.

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 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. The objective is to solve a problem with two variables *x, y*, the constraint

$$100*x + 60*y \leq 600$$

and the three goal constraints

$7*x + 3*y \geq 40$
$10*x + 5*y = 60$
$5*x + 4*y \geq 35$

where the given order corresponds to their priorities.

To increase readability, the model 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 GoalCtr

 uses "mmxprs"

 forward procedure preemptive
 forward procedure printsol(i: integer)

 declarations
  NGOALS=3                          ! Number of goals
  x,y: mpvar                        ! Variables
  dev: array(1..2*NGOALS) of mpvar  ! Deviation from goals
  mindev: linctr                    ! Objective function
  goal: array(1..NGOALS) of linctr  ! Goal constraints
 end-declarations

 limit:= 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


 procedure preemptive
(!
  Add successively the goals to the problem and solve it, until all
  goals have been added or a goal cannot be satisfied. This assumes
  that the goals are given ordered by priority.
!)

! Remove (=hide) goal constraint from the problem
  forall(i in 1..NGOALS) 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: ", getsol(x), ", y: ", getsol(y))

    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

  procedure printsol(i:integer)
   declarations
    STypes={CT_GEQ, CT_LEQ, CT_EQ}
    ATypes: array(STypes) of string
   end-declarations

   ATypes:=[">=", "<=", "="]

   writeln(" Goal", strfmt("Target",12), strfmt("Value",9))
   forall(g in 1..i)
    writeln(strfmt(g,5), strfmt(ATypes(gettype(goal(g))),3),
      strfmt(-getcoeff(goal(g)),9),
      strfmt( getact(goal(g)) -getsol(dev(2*g))+getsol(dev(2*g-1)) ,9))

   forall(g in 1..NGOALS)
    if(getsol(dev(2*g))>0) then
     writeln(" Goal(",g,") deviation from target: -", getsol(dev(2*g)))
    elif(getsol(dev(2*g-1))>0) then
     writeln(" Goal(" ,g, ") deviation from target: +",
             getsol(dev(2*g-1)))
    end-if

  end-procedure

end-model
```

This example again uses procedure `sethidden` to remove constraints from the problem that is solved by the optimizer without deleting them in the problem definition. So effectively, the optimizer solves a series of subproblems of the problem stated originally in Mosel.

When running the program, the user will find that the first two goals can be satisfied, but not the third.

Part C

This part presents some advanced uses of Mosel that go beyond the functionality that is typically required for working with this software. Whilst the two previous parts have shown how to work with the Mosel language, this Part introduces Mosel's C interface. The C interface is provided in the form of two libraries.

It may be of special interest to users who want to

integrate models and/or solution algorithms written with Mosel into some larger system

(re)use already existing parts of algorithms written in C

interface Mosel with other software, for instance for graphically displaying results

Part C - Chapter 1: The 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, but the data and parameters used by a model may be modified via files or run time parameters.

C1.1 Basic Tasks

To work with a Mosel model, in the C language or with the command line interpreter, the model always needs to be compiled, then loaded into Mosel, and finally executed. In this section we show how to perform these basic tasks in C.

C1.1.1 Compiling a Model in C

The following example program shows how Mosel is initialized and terminated in C, and how a model file (extension `.mos`) is compiled into a 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, as for all others in this chapter, we only implement rudimentary testing for errors.

```
#include <stdlib.h>
#include "xprm_mc.h"

int main()
{
 if(XPRMinit())                      /* Initialize Mosel */
  return 1;

 if(XPRMcompmod(NULL, "Models/burglar3", NULL, "Knapsack example"))
  return 2;                          /* Compile the model burglar3.mos,
                                        output the file burglar3.bim */
 XPRMfree();                         /* Free Mosel, clear everything */
 return 0;
}
```

C1.1.2 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 as 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 from 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.

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 int result;

 if(XPRMinit())                        /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("Models/burglar3.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))      /* Run the model */
  return 3;

 XPRMfree();                           /* Free Mosel, clear everything */
 return 0;
}
```

C1.2 Parameters

In Part A the concept of parameters in Mosel was 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 Chapter B2 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:

```
int result;

if(XPRMinit())                                        /* Initialize Mosel */
 return 1;

if((mod=XPRMloadmod("Models/prime.bim",NULL))==NULL)   /* Load a BIM file */
 return 2;

if(XPRMrunmod(mod,&result,"LIMIT=500"))                    /* Run the model */
 return 3;
```

C1.3 Accessing Modeling Objects and Solution Values

Using the Mosel libraries we can easily access information on the different modeling objects.

C1.3.1 Accessing Sets

A complete version of a program 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 Chapter B2 but with all output printing removed because we are doing this in C):

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue, setitem;
 XPRMset set;
 int result, type, i, size, first, last;

 if(XPRMinit())                              /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("Models/prime2.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,"LIMIT=500"))     /* Run the model */
  return 3;

 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 6;
 set = rvalue.set;

 size = XPRMgetsetsize(set);                 /* Get the size of the set */
 if(size>0)
 {
  first = XPRMfirstsetndx(set);              /* Get the number of the first index */
  last = XPRMlastsetndx(set);                /* Get the 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");
 }

 XPRMfree();                                 /* Free Mosel, clear everything */
 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 can then enumerate the elements of the set and obtain their respective values.

C1.3.2 Retrieving Solution Values

The following program executes the model Burglar3 (the same as model Burglar2 from Chapter A3 but with all output printing removed) 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("Models/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 objective function value */

type=XPRMfindident(mod,"x",&rvalue);       /* Get the model object 'x' */
if((XPRM_TYP(type)!=XPRM_TYP_MPVAR)||      /* Check the type: */
   (XPRM_STR(type)!=XPRM_STR_ARR))         /* it must be an array of unknowns */
 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;

XPRMfirstarrentry(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("x(%s) : %g\t (item value: %g)\n", XPRMgetelsetval(set, indices[0],
    &itemname)->string, XPRMgetvsol(mod,x), val);
                                          /* Print the solution value */
} while(!XPRMnextarrentry(varr, indices)); /* Get the next index */

XPRMfree();                               /* Free Mosel, clear everything */
return 0;
}
```

The array of variables `x` is enumerated using function `XPRMnextarrentry`. This function 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 modified). This function systematically runs through all possible combinations of index-tuples, hence the hint at dense arrays in the example. In the case of sparse arrays it is preferable to use a different enumeration function that only enumerates those entries that are defined (see next section).

C1.3.3 Sparse Arrays

In Chapter A5 the Transport problem was introduced. The objective of this problem is to calculate the flows $flow_{pr}$ 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 $flow_{pr}$ are therefore defined as a sparse array. The following example prints out all existing entries of the array of variables.

```
#include <stdio.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("Models/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 unkn owns */
  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  */
 XPRMfirstarrtruentry(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(!XPRMnextarrtruentry(varr,indices));   /* Get next true index tuple*/
 printf("\n");

 free(sets);
 free(indices);

 XPRMfree();                             /* Free Mosel, clear everything */
 return 0;
}
```

C1.4 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 Xpress-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
  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
  WTMAX=102                        ! Max weight allowed
  x: array(Items) of mpvar         ! 1 if we take item i; 0 otherwise
 end-declarations

 !  Item:  ca   ne  va  pi  tv  vi  ch  br
 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)*x(i)

 ! Weight restriction
 WtMax:= sum(i in Items) WEIGHT(i)*x(i) <= WTMAX

 ! All x are 0/1
 forall(i in Items) x(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 reads in the Mosel model and solves the problem by direct calls to the 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 that is provided by module `mmxprs`, we first need to obtain the reference of this library (by using function `XPRMfinddso`).

```
#include <stdio.h>
#include "xprm_rt.h"
#include "xprs.h"

int main()
{
 XPRMmodel mod;
 XPRMdsolib dso;
 XPRSprob prob;
 int result, ncol, len, i;
 double *sol, val;
 char *names;

  if(XPRMinit())                         /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("Models/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 Xpress-Optimizer */
 if((dso=XPRMfinddso("mmxprs"))==NULL)
  return 4;
 if(XPRMgetdsoparam(mod, dso, "xprs_problem", &result, (XPRMalltypes *)&prob))
  return 5;

 if(XPRSmaxim(prob, "g"))               /* Solve the problem */
  return 6;
```

```
  if(XPRSgetintattrib(prob, XPRS_MIPSTATUS, &result))
   return 7;
                                        /* Test whether a solution is found */
  if((result==4) || (result==6))
  {
   if(XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &val))
    return 8;
   printf("Objective value: %g\n", val); /* Print the objective function value */
   if(XPRSgetintattrib(prob, XPRS_COLS, &ncol))
    return 9;
   if((sol = (double *)malloc(ncol * sizeof(doub le)))==NULL)
    return 10;
   if(XPRSgetsol(prob, sol, NULL, NULL, NULL))
    return 11;                          /* Get the primal solution values */
   if(XPRSgetintattrib(prob, XPRS_NAMELENGTH, &len))
    return 11;                          /* Get the max imum name length */
   if((names = (char *)malloc((len*8+1)*ncol*sizeof(char)))==NULL)
    return 12;
   if(XPRSgetnames(prob, 2, names, 0, ncol -1))
    return 13;                          /* Get the variable names */
   for(i=0; i<ncol; i++)                / * Print out the solution */
     printf("%s: %g\n", names+((len*8+1)*i), sol[i]);
   free(names);
   free(sol);
  }

  XPRMfree();                           /* Free Mosel, clear everything */
  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. Nevertheless, if this mode 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 back to Mosel.

Index