Introduction, Xpress overview
» Introduction to Xpress

» Modeling with Mosel:
  » Linear and Mixed Integer Programming (LP and MIP)
  » Accessing data sources
  » Programming language features

» Embedding models in applications
At the end of the course you will

- be familiar with optimization methods and the terminology used to describe them
- be confident about formulating optimization models and understanding the solution
- know how to use Xpress to model and solve problems
- be able to embed a model in an application
» Not a replacement for the reference manuals!
» Focuses on areas that are of practical importance
» Does not try to be exhaustive
» Pointers to reference material at the end of every chapter
Overview of Xpress
Overview of Xpress

» Optimization algorithms
  » enables you to solve different classes of problems
  » built for speed, robustness and scalability

» Modeling interfaces
  » enables you to provide your problem in the most suitable way for your application
  » built for ease of use and interfacing
Modeling interfaces

» Mosel
  » formulate model and develop optimization methods using Mosel language / environment

» BCL
  » build up model in your application code using object-oriented model builder library

» Optimizer
  » read in matrix files
  » input entire matrix from program arrays
Mosel

» A modeling and solving environment
  » integration of modeling and solving
  » programming facilities
  » open, modular architecture

» Interfaces to external data sources (e.g. ODBC, host application) provided

» Language is concise, user friendly, high level

» Best choice for rapid development and deployment
Mosel: Components and interfaces

» Mosel language: to implement problems and solution algorithms
⇒ model or Mosel program

» Mosel Model Compiler and Run-time Libraries: to compile, execute and access models from a programming language
⇒ C/C++, C#, Java, or VB program
» Mosel Native Interface (NI): to provide new or extend existing functionality of the Mosel language
⇒ module

» Xpress-IVE: graphical user interface, representation of the problem matrix, solution status/progress graphs, and result display
Mosel Libraries

» Embed Mosel models directly in your application
» Access the solution within your application
» Compiled models are platform independent
» Enjoy benefits of structured modeling language and rapid deployment when building applications
» Available for C, Java, C#, and VB
» Visual Studio style visual development environment for optimization & model building with Mosel
» Mosel model editor & compiler
» Real time graphs show optimization performance
» Browse solution values in entity tree
Mosel and Optimizer Consoles

» Stand-alone command line executables with text interfaces
» Useful for simple deployment using batch/script files
» Available for all platforms supported by Xpress
Why use modeling software?
Why use modeling software?

- Problem conception
- Interpretation & analysis

Model

- Computational problem instance
- Computational solution instance

Model solution

- Human
- Computer
Why use modeling software?

» Developing a working model is the difficult bit
» Important to have software that helps
  » speed to market
  » verify correctness
  » maintenance & modification
  » algorithmic considerations
  » execution speed
» The concepts we describe – how to formulate and solve problems – apply to all modeling software

» In this course we will use the Xpress-IVE development environment with the Xpress-Mosel language because it is
  » easy to understand and learn
  » easy to use
Xpress-IVE demonstration
Xpress-IVE demonstration

» Models: new, saving, opening, switching
  - start a new model
  - open an existing model
  - save current model
  - show list of available modules

» Bars: editor, entity, info, output (run)
  - switch between window layouts
Xpress-IVE demonstration

» Editor: colors, auto-complete, tool tips

- copy selection
- cut selection
- paste selection
- go to next / last line with same indentation
- go to previous / next cursor position (line)
- undo / redo last editor command
» Compile, run
  • compile current model
  • execute current model
  • open run options dialog
  • pause execution
  • interrupt execution
  • search for the N best solutions
  • start infeasibility repair
» Output bar: log, stats, matrix, graphs, tree
» Viewing solution values
» Problem and matrix export and import
 生成 BIM 文件
导出问题矩阵
优化导入的矩阵
Xpress-IVE demonstration

» Search, bookmark
  - search
  - delete bookmarks

» Help
  - help
  - model generation wizzard & example models
  - module generation wizzard
Xpress-IVE demonstration

» Debugger
  - set/delete breakpoint at cursor
  - define conditional breakpoint
  - start/stop debugger
  - step over an expression
  - step into an expression
  - run up to the cursor
  - show debugger options dialog

» Profiler
  - start the profiler
Reference material

» The manual *Getting Started with Xpress* introduces first time or occasional users to modeling with Mosel and BCL, or the direct Optimizer interface

» The *Evaluators Guide and Advanced Evaluators Guide* provide a quick walk-through of the Getting Started examples and some more advanced features
Modeling with Mosel
Overview

» Modeling basics
» Accessing data sources
» Advanced modeling topics
» Programming language features
» Mosel modules and packages
Modeling basics

» A first model
» Data structures and loops
» Model building style
» Definition of decision variables and constraints
» Solving with Xpress-Optimizer
» Solution output
Example: Chess problem

» A joinery makes two different sizes of boxwood chess sets.

» The small set requires 3 hours of machining on a lathe, and the large set requires 2 hours. There are 4 lathes with skilled operators who each work a 40 hour week.

» The small chess set requires 1 kg of boxwood, and the large set requires 3 kg. Only 200 kg of boxwood can be obtained per week.
Example: Chess problem

» Each of the large chess sets yields a profit of $20, and one of the small chess sets has a profit of $5.

» How many sets of each kind should be made each week so as to maximize profit?
Chess problem: Mathematical formulation

\[
x_l - \text{quantity of large chess sets made}
x_s - \text{quantity of small chess sets made}
\]

\[
\begin{align*}
\max \quad & z = 5 \cdot x_s + 20 \cdot x_l \\
\text{s.t.} \quad & 3 \cdot x_s + 2 \cdot x_l \leq 160 (= 4 \cdot 40) \quad \text{(lathe time)} \\
& x_s + 3 \cdot x_l \leq 200 \quad \text{(wood)} \\
& x_s, x_l \geq 0
\end{align*}
\]
Chess problem: Graphical solution

The diagram illustrates the optimal solution for a given problem within a graphical context. The axes represent different variables, and the shaded area indicates the feasible region for the solution. The optimal solution lies at the intersection of the constraints, marked as "optimal solution."
model "Chess 1"
uses "mmxprs"

! Use Xpress-Optimizer for solving

declarations
  xs: mpvar
  xl: mpvar
end-declarations

3*xs + 2*xl <= 160  ! Constraint: limit on working hours
xs + 3*xl <= 200    ! Constraint: raw mat. availability

maximize(5*xs + 20*xl)  ! Objective: maximize total profit

end-model
model "Chess 1"
    ...
end-model
uses statement: Say we will use the Xpress-Optimizer library, so that we can solve our problem

Options:
» noimplicit: force all objects to be declared
» explterm: Use ‘;’ to mark line ends

uses 'mmxprs'
options noimplicit
options explterm
» mpvar means mathematical programming variable or decision variable

» Decision variables are unknowns: they have no value until the model is run, and the optimizer finds values for the decision variables
» In optimization problems, decision variables are often just called variables
» In computer programs, a variable can be used to refer to many different types of objects
» For instance, in Mosel models, a program variable can be used to refer to a decision variable, as well as integers, reals, etc.
Bounds on decision variables

» Variables can take values between 0 and infinity by default
» Other bounds may be specified

\[ x \leq 10 \]
\[ y(1) = 25.5 \]
\[ y(2) \text{ is_free} \]
\[ z(2,3) \geq -50 \]
\[ z(2,3) \leq 50 \]
Constrains

> Have type \texttt{linctr} – linear constraint

declarations
  Wood: \texttt{linctr}
  Inven: \texttt{array(1..10) of linctr}
end-declarations
Constraints

» The ‘value’ of a constraint entity is a linear expression of decision variables, a constraint type, and a constant term

» Set using an assignment statement

\[ \text{Wood} := x_s + 3 \times x_1 \leq 200 \]
An objective function is just a constraint with no constraint type

```plaintext
declarations
    MinCost: linctr
end-declarations

MinCost := 10*x(1) + 20*x(2) + 30*x(3) + 40*x(4)
```
» Generate the matrix and solve the problem:

minimize(MinCost)
maximize(5*xs + 20*x1)

» Load the matrix:

loadprob(MinCost)

» Matrix export:

exportprob(0, "explout", MinCost)
» Can access and manipulate the solution values within the model

```cpp
writeln('Solution: ', getobjval)
writeln('xs = ', getsol(xs))
writeln('xl = ', getsol(xl))
write('Wood: ', getact(Wood), ', ', getact(Wood), ' ')
writeln(getslack(Wood))
```

» Solution values of constraints
activity value + slack value = RHS
» Execute the model \texttt{chess1.mos}.
» Add printing of the solution values.
» Is the solution realistic/desirable?
» Constrain the variables to take integer values only.
» Add output of constraint activity and slack values.
» Executing model `chess1.mos` with IVE:
  » double click on the model file to start IVE or open the file from within IVE
  » click on the run button: ➤

» Model execution from the command line:

```
mosel -c "exe chess1.mos"
```

» or:

```
mosel
exe chess1.mos
quit
```
Solution:
Completed model Chess 1

model "Chess 1 (completed)"
uses "mmxprs"

declarations
xs,xl: mpvar
end-declarations

Profit:= 5*xs + 20*xl
Time:= 3*xs + 2*xl <= 160
Wood:= xs + 3*xl <= 200
xs is_integer; xl is_integer

maximize(Profit)

writeln("Solution: ", getobjval)
writeln("xs: ", getsol(xs), " xl: ", getsol(xl))
write("Time: ", getact(Time))
writeln(" ", getslack(Time))
end-model
» What happens if machines operate 35 instead of 40 hours?

» Calculate spare capacity: `getslack`, `getactivity`
What is the cost of an extra unit of wood/extra working hour?

Reduced cost: `getrcost`
» What is the cost of producing an additional unit of each product?

» Dual values (‘shadow prices’): `getdual`

» Increase price of `xl` to reach break even point
Solution analysis

» Limit the amount of $x_l$. 
uses "mmxprs"
options explterm ! Use ';;' to mark line ends

dependencies

Allvars: set of mpvar; ! Set of variables
DescrV: array(Allvars) of string; ! Descriptions of variables
xs, xl: mpvar;
end-declarations

DescrV(xs):= "Small"; DescrV(xl):= "Large";

Profit:= 5*xs + 20*xl; ! Objective function
Time:= 3*xs + 2*xl <= 160; ! Constraints
Wood:= xs + 3*xl <= 200;
xs is_integer; xl is_integer;

maximize(Profit);
writeln("Solution: ", getobjval);
forall(x in Allvars) writeln(DescrV(x), ": ", getsol(x));
Data structures

» **Set**: unordered collection of objects of the same type
  » used as index sets
  » special type range sets (= interval of integers)

» **Array**: multidimensional table of objects of the same type
  » used for data, decision variables, constraints
  » may be dynamic or static
uses "mmxprs"

declarations
  R = 1..2  ! Index range
  DUR, WOOD, PROFIT: array(R) of real  ! Coefficients
  x: array(R) of mpvar  ! Array of variables
end-declarations

DUR :: [3, 2]  ! Initialize data arrays
WOOD :: [1, 3]
PROFIT :: [5, 20]

sum(i in R) DUR(i) * x(i) <= 160  ! Constraint definition
sum(i in R) WOOD(i) * x(i) <= 200
forall(i in R) x(i) is_integer
maximize(sum(i in R) PROFIT(i) * x(i))
writeln("Solution: ", getobjval)
declarations
NWEEKS = 20 ! Integer constant
DATA_DIR = 'c:/data' ! String constant
NPROD: integer ! Integer variable
SCOST: real ! Real variable
DIR: string ! String variable
IF_DEBUG: boolean ! Boolean variable

PRODUCTS = {"P1", "P2", "P4"} ! Constant set of string
S: set of integer ! Variable set of integer
R: range ! Range of integers
COST: array(1..3,1..4) of real ! Array of real
end-declarations
Data initialization

NPROD:= 50
SCOST:= 5.4
DIR:= 'c:/data'
IF_DEBUG:= true

S:= {10, 0, -5, 13}
R:= 1..NPROD
COST:: [11, 12, 13, 14,
      21, 22, 23, 24,
      31, 32, 33, 34]
» Sum up an array of variables in a constraint

Ctr1:= \( \sum_{p=1}^{10} (\text{RES}(p) \times \text{buy}(p) + \text{sell}(p)) \) \(\leq 100\)

Ctr2:= \( \sum_{p \in \text{PRODUCTS}} (\text{buy}(p) + \sum_{r=1}^{5} \text{make}(p,r)) \) \(\leq 100\)

Ctr3:= \( \sum_{p=1}^\text{NP} \left( 2 \times \text{CAP}(p) \times \text{buy}(p)/10 + \text{SCAP}(p) \times \text{sell}(p) \right) \) \(\leq \text{MAXCAP}\)
Loops

» Use a loop to assign an array of constraints

\[
\text{forall } (t \text{ in } 2..NT) \\
\text{Inven}(t) := \text{bal}(t) = \text{bal}(t-1) + \text{buy}(t) - \text{sell}(t)
\]
» Use `do/..end-do` to group several statements into one loop

```plaintext
forall(t in 1..NT) do
    MaxRef(t) := sum(i in PRODUCTS)
        use(i,t) <= MAXREF(t)

    Inven(t) := store(t) = store(t-1) + buy(t) - use(t)
end-do
```
» Can nest `forall` statements

```plaintext
forall(t in 1..NT) do
    MaxRef(t) := sum(i in 1..NI) use(i,t) <= MAXREF(t)

forall(i in 1..NI)
    Inven(i,t) := store(i,t) = store(i,t-1) + buy(i,t) - use(i,t)
end-do
```
May include conditions in sums or loops

\[
\text{forall}(c \text{ in } 1..10 \mid \text{CAP}(c) \geq 100.0)
\]
\[
\text{MaxCap}(c) :=
\]
\[
\text{sum}(i \text{ in } 1..10, j \text{ in } 1..10 \mid i \neq j)
\]
\[
\text{TECH}(i,j,c) \times x(i,j,c) \leq \text{MAXTECH}(c)
\]
» Can extend over several lines and use spaces
» However, a line break acts as an expression terminator
» To continue an expression, it must be cut after a symbol that implies continuation (e.g. + - , )
You should aim to build a model with sections in this order:

- **Constant data**: declare, initialize
- **All non-constant objects**: declare
- **Variable data**: initialize / input / calculate
- **Decision variables**: create, specify bounds
- **Constraints**: declare, specify
- **Objective**: declare, specify, optimize
In both LP and MIP it is very important to distinguish between known values data, parameters, etc. and unknown values decision variables.

All constraints must be linear expressions of the variables.
Suggestion: name objects as follows

- known values (data) using upper case
- unknown values (variables) using lower case
- constraints using mixed case

so that it is easy to distinguish between them, and see that constraints are indeed linear.
Model building style

» Variables are actions that your model will prescribe
» Use verbs for the names of variables
  » this emphasizes that variables represent ‘what to do’ decisions
» Indices are the objects that the actions are performed on
» Use nouns for the names of indices
» Using named index sets/ranges
  » improves the readability of a model
  » makes it easier to apply the model to different sized data sets
  » makes the model easier to maintain
  » may speed up your model
Model building style

» Try to include ‘Min’ or ‘Max’ in the name of your objective function

» An objective function called ‘Obj’ is not very helpful when taken out of context!
Comments are essential for a well written model
Always use a comment to explain what each parameter, data table, variable, and constraint is for when you declare it
Add extra comments to explain any complex calculation etc.
Comments in Mosel:

declarations

PRODUCTS = 1..NP  ! Set of products
TIMES = 1..NT      ! Set of time periods
make: array(PRODUCTS, TIMES) of mpvar
      ! Amount of p produced in time t
sell: array(PRODUCTS, TIMES) of mpvar
      ! Amount of p sold in time t

end-declarations

(! And here is a multi-line
comment !)   forall(t in TIMES)
Accessing data sources

» The initializations block
» Dynamic arrays
» Run-time parameters
» Using other data sources
» Text files
» ODBC
» Sparse data
Separation of problem logic and data

» Typically, the model logic stays constant once developed, with the data changing each run

» Editing the model can create errors, expose intellectual property, and is impractical for industrial size data

» It makes good sense to fix the model and obtain data from their source
uses "mmxprs"

declarations
  PRODS = 1..2  ! Index range
  DUR, WOOD, PROFIT: array(PRODS) of real  ! Coefficients
  x: array(PRODS) of mpvar  ! Array of variables
end-declarations

initializations from "chess.dat"  ! Read data from file
  DUR WOOD PROFIT
  ! chess.dat: PROFIT: [5 20]
  ! DUR: [3 2]
  ! WOOD: [1 3]
end-initializations

sum(i in PRODS) DUR(i)*x(i) <= 160  ! Constraint definition
sum(i in PRODS) WOOD(i)*x(i) <= 200
forall(i in PRODS) x(i) is_integer
maximize(sum(i in PRODS) PROFIT(i)*x(i))
writeln("Solution: ", getobjval)
Every data item/table has a label, its identifier

Single line comments (marked with ‘!’)

Data file for ‘chess4.mos’

DUR: [3 2]
WOOD: [1 3]
PROFIT: [5 20]
Sparse data format

» Every data entry specified with its index tuple
» Can read data from one labeled data source into several Mosel data tables at once
  » data tables must have identical indices

initializations from 'chess.dat'
  [DUR, WOOD, PROFIT] as 'ChessData'
end-initializations
» Format of data file with several data values in one labeled data range (use a * for a missing data value)

! chess.dat

ChessData: [
  (1) [3 1 5]
  (2) [2 3 20]
]
» You can write out values in an analogous way to reading them in using **initializations** to

» To write out the solution values of variables, or other solution values (slack, activity, dual, reduced cost) you must first put the values into a data table
Writing data out to text files

declarations
  x_sol: array(PRODS) of real
end-declarations

forall(i in PRODS)
  x_sol(i) := getsol(x(i))

initializations to 'result.dat'
  x_sol
end-initializations
fopen("result.dat", F_OUTPUT+F_APPEND)

forall(i in PRODS)
  writeln(i, ": ", getsol(x(i))

fclose(F_OUTPUT)
Modify the model *chess4.mos* to use indices of type *string*.

Execute this new model *chess4s.mos* with data set *chess2.dat*.

Output the solution values to file *sol.dat* using initializations to.

Modify the models further to read the contents of the index set from file (*chess5.mos, chess5s.mos*).
Dynamic arrays

» Mosel provides a user friendly and efficient means of modeling mathematical programming problems

» Objects such as dynamic arrays and variable index sets, together with efficient loops and sums, allow large scale models to be written easily, and execute quickly.
Dynamic arrays

» Dynamic array: indexing sets not known at declaration, or array explicitly marked dynamic

» Initialize dynamic data arrays from text files or using ODBC
  » data must use sparse format
  » this is so Mosel can work out the values of the indices
  » reading in the data array initializes both the index values and the data values at the same time
» An entry of a dynamic array is only created when a value is assigned to it
» Decision variables don’t get created, because you don’t assign values to them
» To create decision variables in a dynamic array, use the `create` procedure
Dynamic arrays of decision variables

declarations

TIME: range ! = set of contiguous integers
COST: array(TIME) of real
use: array(TIME) of mpvar

end-declarations

(...)

! Read in COST data etc

forall(t in TIME | exists(COST(t)))
  create(use(t))
Dynamic arrays of decision variables

» Note: if you declare decision variables after reading in the data, then decision variables will be created for all combinations of the index set elements that exist at that time

» Do not use create in this case

» Define decision variables before reading in data if you want to use create to control exactly which elements get created
» Use dynamic arrays
  » to size data tables automatically when the data is read in
  » to initialize the index values automatically when the data is read in
  » to conserve memory when storing sparse data
  » to eliminate index combinations without using conditions each time
Dynamic arrays

» Don’t use dynamic arrays
  » when you can use an ordinary (static) array instead
  » when storing dense data, and you can size the data table and initialize the indices in some other way (dynamic arrays are slower and use more memory than a static array when storing dense data)
uses "mmxprs"
parameters
  FILENAME="chess.dat" ! Name of the data file
end-parameters

declarations
  PRODS = 1..2 ! Index range
  DUR, WOOD, PROFIT: array(PRODS) of real ! Coefficients
  x: array(PRODS) of mpvar ! Array of variables
end-declarations

initializations from FILENAME ! Read data from file
  DUR WOOD PROFIT
end-initializations

sum(i in PRODS) DUR(i)*x(i) <= 160 ! Constraint definition
sum(i in PRODS) WOOD(i)*x(i) <= 200
forall(i in PRODS) x(i) is_integer
maximize(sum(i in PRODS) PROFIT(i)*x(i))
Run-time parameters

Parameters
  a special type of constant
  default value may be overridden at run-time

parameters
  DATA_DIR = 'c:/data'
  DEBUG = true
  NUM_RECORDS = 1000
end-parameters
Run-time parameters

» The value in the model is used by default

» A different value may be given at run-time
  » In IVE, an alternative value may be set in the Build ➤ Options dialogue
  » When running a Mosel model from an application, an alternative value can be set in the parameters string
Run-time parameters

» A **parameters** section must come at the top of the model
» after any **uses** or **options** statements
» before any other statements
Parameters are especially useful for passing directories/paths into the model

- all files referenced in the model should use a directory parameter
- otherwise, Mosel may not be able to find the file when the model is deployed (the default path differs when run from an application)
- use ‘+’ to join strings
Run-time parameters

» Specifying directory paths
  » preferably use ‘/’ as directory separator

```plaintext
parameters
  DIR = '.'
end-parameters

fopen(DIR+’/cap.dat’, F_INPUT)
...
fclose(F_INPUT)
...
initializations from DIR+’/cost.dat’
...
» In models `chess5.mos` and `chess5s.mos` turn the data file name into a run-time parameter.

» Re-run your model `chess5s.mos` with the larger data set `chess3.dat` without changing the filename in the model.
Setting runtime parameters within IVE:
- select menu *Build* ➤ *Options* or click on the button
- check *Use model parameters* to activate the parameter input field and enter the new value(s)

Runtime parameters from the command line:

```
mosel -c "exe chess5s.mos DATAFILE='chess3.dat'"
```

or:

```
mosel
exe chess5s.mos DATAFILE='chess3.dat'
quit
```
Using other data sources

» The initializations block can work with many different data sources and formats thanks to the notion of I/O drivers
» I/O drivers for physical data files: mmodbc.excel, mmoci.oci, mmetc.diskdata
» Other drivers available, e.g. for data exchange in memory
» Change of the data source = change of the I/O driver, no other modifications to your model
First, must check ODBC driver for your chosen data source (external to Xpress)

» Start » Settings » Control Panel » Administrative Tools » Data Sources (ODBC)

» Check that data source is defined, and note its name (the data source name, DSN)
Next, identify specific data source – a database or spreadsheet
  - note its location (path)
  - the data must be in a table in a database, or a named range in a spreadsheet
Now, in your model

- use the `mmodbc` module (requires licence)
- use the `odbc` driver in `initializations` blocks, or
- write out the corresponding SQL commands:
  - set up an ODBC data connection to the specific data source
  - input data using SQL statements
  - disconnect
Reading data via ODBC

» Excel spreadsheet ('ChessData' = range in the spreadsheet):

initializations from 'mmodbc.odbc:chess.xls'
[DUR, WOOD, PROFIT] as 'ChessData'
end-initializations

» Access database ('ChessData' = data table):

initializations from 'mmodbc.odbc:debug;chess.mdb'
[DUR, WOOD, PROFIT] as 'ChessData'
end-initializations
Data export to a database

initializations to 'mmodbc.odbc:debug;chess.mdb'
  x_sol as 'ChessSol'
end-initializations

» Before every new run, delete the data from the previous run in the destination range/table
» Otherwise the new results will either be appended to the existing ones or, if ‘PRODS’ has been defined as key field in a database, the insertion will fail
Special notes for data export to Excel

» Make sure the ‘Read Only’ option is disabled in the ODBC data source set-up options

» Define the destination range in the spreadsheet, with one line of column headings, one line of dummy data, and no other data

» Excel does not support the full range of ODBC functionality (commands like ‘update’ or ‘delete’ will fail)

⇒ preferably use direct connection (excel driver)
Data exchange with MS Excel

» Software-specific driver excel for MS Excel
  » use *mmodbc* module (requires licence)
  » use the *excel* driver (instead of *odbc*) in *initializations blocks*
  » no driver setup required (works with standard Excel installation)
  » simply replace "*mmodbc.odbc:*" by "*mmodbc.excel:skiph;*" in the preceding examples
Data exchange with Oracle

» Software-specific driver oci for Oracle databases
  » use mmoci module (requires licence)
  » setup: Oracle’s Instant Client package must be installed on the machine running the Mosel model
  » in initializations blocks replace "mmodbc.odbc:" by "mmoci.oci:" in the preceding examples
  » supports SQL statements (replace the prefix SQL by OCI)
» The I/O driver odbc generates automatically the SQL commands required to connect to the database/spreadsheet
» For advanced uses module mmodbc also defines most standard SQL commands directly for the Mosel language
» Check that the ODBC DSN for Excel is set up on your computer

» Re-run your model chess5.mos with the Excel file chess.xls
We have seen that it is possible to completely separate the data and the model.

The model specifies the logic of the problem, without any reference to its size.

The model can be applied to any data instance, simply by providing data files.
» Refer to the Mosel User Guide for a detailed introduction to working with Mosel.
» The book Applications of optimization with Xpress-MP provides a large collection of examples models from different application areas.
» See the whitepaper Using ODBC and other database interfaces with Mosel for further detail on data handling.
Advanced modeling topics

» MIP variable types
» Modeling with binary variables
» MIP variable types
» Modeling with binary variables
Binary variables
- can take either the value 0 or the value 1 (do/ don’t do variables)
- model logical conditions

\( x(4) \text{ is_binary} \)
MIP variable types

» Integer variables
  » can take only integer values
  » used where the underlying decision variable really has to take on a whole number value for the optimal solution to make sense

\( x(7) \text{ is\_integer} \)
Partial integer variables

- can take integer values up to a specified limit and any value above that limit
- 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

\[ x(1) \text{ is\_partint 5 ! Integer up to 5, then continuous} \]
MIP variable types

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

\[ x(2) \text{ is-semcont 6 } \quad ! \text{A 'hole' between 0 and 6, then continuous} \]
MIP variable types

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

\[ x(3) \text{ is}_\text{semint} 7 \] ! A 'hole' between 0 and 7, then integer
MIP variable types

» Special Ordered Sets of type one (SOS1)
  » an ordered set of variables at most one of which can take a non-zero value
  » single choice among several possibilities

» Special Ordered Sets of type two (SOS2)
  » 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
  » e.g. approximation of non-linear functions with a piecewise linear function
SOS definition

» WEIGHT array determines the ordering of the variables:

\[ MYSOS := \sum_{i \in IRng} \text{WEIGHT}(i) \times x(i) \text{ is sosX} \]

where is sosX is either is sos1 or is sos2
» Alternative: set $S$ of set members, linear constraint $L$ with ordering coefficients (= reference row entries):

$\text{makesos1}(S,L); \text{makesos2}(S,L)$

» must be used if the coefficient $\text{WEIGHT}(i)$ of an intended set member is zero

» Note: the ordering coefficients must all be distinct (or else they are not doing their job of supplying an order!)
» Projects A, B, C, D
» Binary variables \(a, b, c, d\)
   » do at most 3 projects: \(a + b + c + d \leq 3\)
   » must do D if A done: \(d \geq a\)
   » can only do C if both A and B done:
     \(c \leq (a + b) / 2\)
     \(c \leq a, c \leq b\)
» Either

\[ 5 \leq x \leq 10 \]

or

\[ 80 \leq x \leq 100 \]
Disjunctions

Introduce a new variable:

\[ \text{ifupper} : 0 \text{ if } 5 \leq x \leq 10; \quad 1 \text{ if } 80 \leq x \leq 100 \]

\[ x \leq 10 + (100 - 10) \cdot \text{ifupper} \quad [1] \]

\[ x \geq 5 + (80 - 5) \cdot \text{ifupper} \quad [2] \]
Disjunctions

» Either \( 5 \leq \sum_{i} A_i x_i \leq 10 \)
  or \( 80 \leq \sum_{i} A_i x_i \leq 100 \)

\[
\sum_{i} A_i x_i \leq 10 + 90 \cdot \text{ifupper}
\]

\[
\sum_{i} A_i x_i \geq 5 + 75 \cdot \text{ifupper}
\]
Absolute values

» Two variables

with

\[ x_1, x_2 \]

want

\[ y = |x_1 - x_2| \]

\[ 0 \leq x_i \leq U \quad [1. \ i] \]
Absolute values

» Introduce binary variables

\[ d_1, d_2 \]

to mean
\[ d_1 : 1 \text{ if } x_1 - x_2 \text{ is the positive value} \]
\[ d_2 : 1 \text{ if } x_2 - x_1 \text{ is the positive value} \]
Absolute values

MIP formulation of $y = |x_1 - x_2|$

1. $0 \leq x_i \leq U$ \hspace{1cm} [1.i]
2. $0 \leq y - (x_1 - x_2) \leq 2 \cdot U \cdot d_2$ \hspace{1cm} [2]
3. $0 \leq y - (x_2 - x_1) \leq 2 \cdot U \cdot d_1$ \hspace{1cm} [3]
4. $d_1 + d_2 = 1$ \hspace{1cm} [4]
Take a look at the capital budgeting model in `capbgt.mos`: the objective is to determine the most profitable choice among 8 possible projects, subject to limited resources (personnel and capital).

Formulate the following additional constraints:

- P1 can only be done if P2 is done
- P1 can only be done if P3 and P6 are done
- It is not possible to do both P5 and P6
- Either P1 and P2 must be done or P3 and P4 (but not both pairs).
Solution

! p1 can only be done if p2 is done
\[ x(2) \geq x(1) \]

! p1 can only be done if p3 and p6 are done
\[ \frac{x(3) + x(6)}{2} \geq x(1) \]

! It is not possible to do both p5 and p6
\[ x(5) + x(6) \leq 1 \]

! Either p1 and p2 must be done or p3 and p4 (but not both pairs).
\[ x(1) = x(2); \ x(3) = x(4) \]
\[ x(1) + x(2) = 2 - (x(3) + x(4)) \]
Programming language features

» Selections
» Loops
» Functions and procedures
» Data structures
» Programming solution algorithms
Mosel: A programming environment

» Selections
» Loops
» Set operations
» Subroutines
» Data structures
Selections

```plaintext
if

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

case

case A of
    -1000..10 : x >= 35
    20..1000 : x <= 7
    12, 15 : x = 1
    else
        x = 0
end-case
```
Loops

» forall [do]
» while [do]
» repeat until
Example: Prime numbers

» Implements the ‘Sieve of Eratosthenes’.

\[ SNumbers = \{2, \ldots, L\} \]
\[ n := 2 \]
repeat
\[ \text{while } (n \notin SNumbers) \quad n := n + 1 \]
\[ SPrime := SPrime \cup \{n\} \]
\[ i := n \]
\[ \text{while } (i \leq L) \]
\[ SNumbers := SNumbers \setminus \{i\} \]
\[ i := i + n \]
until \[ SNumbers = \{\} \]
Example: Prime numbers

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, ":")
Example: Prime numbers

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
Operations on sets

Set operators include

- union: +
- intersection: *
- difference: -

Logical expressions using sets include

- subset: Set1 <= Set2
- superset: Set1 >= Set2
- equals: Set1 = Set2
- not equals: Set1 <> Set2
- element of: 'Oil5' in Set1
- not element of: 'Oil5' not in Set1
Functions and procedures

» Similar structure as model, including the declarations blocks

» Terminated by end-function or end-procedure

» Function defines returned with its return value

» forward declaration

» Overloading possible (each version with a different number or types of arguments)
1. Choose a middle value $v$ for partitioning (here: $v = (\text{min} + \text{max}) / 2$)
2. Divide the list into two parts ‘left’ (all elements $x < v$) and ‘right’ (all elements $x > v$)
3. Repeat from 1. for lists ‘left’ and ‘right’
Example: Quick Sort algorithm

model "Quick Sort"
parameters
    LIM=50
end-parameters

forward procedure qsort(L:array(range) of integer)
forward procedure qsort(L:array(range) of integer, s,e:integer)

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

forall(i in 1..LIM) T(i):=round(.5+random*LIM)
writeln(T)
time:=gettime

qsort(T)  ! Sort the array
writeln(T) ! Print the sorted array
Example: Quick Sort algorithm

! 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

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

! Sorting routine

procedure qsort(L:array(range) of integer, s,e:integer)
  v:=L((s+e) div 2)
i:=s; j:=e
repeat
  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
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

end-model
Data structures

» array
» set
Data structures

» array
» set
» list
» record
Data structures

» array
» set
» list
» record

» ... and any combinations thereof, e.g.,

S: set of list of integer
A: array(range) of set of real
List

» Collection of objects of the same type
» May contain the same element several times
» Order of list elements is specified by construction

« Handling: cuthead, splittail, reverse...

declarations

L: list of integer
M: array(range) of list of string

end-declarations

L:= [1, 2, 3, 4, 5]
M:: (2..4)[['A','B','C'], ['D','E'], ['F','G','H','I']]
» Finite collection of objects of any type
» Each component of a record is called a ‘field’ and is characterized by its name and its type

```plaintext
declarations
ARC: array(ARCSET:range) of record
  Source, Sink: string  ! Source and sink of arc
  Cost: real     ! Cost coefficient
end-record
end-declarations

ARC(1).Source := "B"
ARC(3).Cost := 1.5
```
User types

» Treated in the same way as the predefined types of the Mosel language

» New types are defined in declarations blocks by specifying a type name, followed by =, and the definition of the type

```plaintext
declarations
  myreal = real
  myarray = array(1..10) of myreal
  COST: myarray
end-declarations
```
User types

» Typical uses
  » shorthand for repetitions in declarations
  » naming records

declarations
  arc = record
    Source,Sink: string    ! Source and sink of arc
    Cost: real            ! Cost coefficient
  end-record
  A: arc
  ARC: array(ARCSET:range) of arc
end-declarations
Summary: Language features

» Data structures: array, set, list, record

» Selections: if-then-[elif-then]-[else], case

» Loops: forall-[do], while-[do], repeat-until

» Operators:
  » standard arithmetic operators
  » aggregate operators (sum, prod, and, or, min, max, union, intersection)
  » set operators

» Subroutines: functions, procedures (forward declaration, overloading)
Mosel: A solving environment

- No separation between ‘modeling statements’ and ‘solving statements’
- Programming facilities for pre/postprocessing, algorithms
- Principle of incrementality
- Not solver-specific
- Possibility of interaction with solver(s)
Solving: Variable fixing heuristic

» Solution heuristic written with Mosel
» Program split into several source files
Solving: Variable fixing heuristic (main file)

```mos
model Coco
    uses "mmxprs"

    include "fixbv_pb.mos"
    include "fixbv_solve.mos"

    solution:=solve
    writeln("The objective value is: ", solution)

end-model
```
declarations
RF=1..2               ! Range of factories (f)
RT=1..4               ! Range of time periods (t)
(...)
openm: array(RF,RT) of mpvar
end-declarations

(...)
forall(f in RF,t in 1..NT-1) Closed(f,t):= openm(f,t+1) <= openm(f,t)
forall(f in RF,t in RT) openm(f,t) is_binary
function solve: real
  declarations
    osol: array(RF, 1..2) of real
    bas: basis
  end-declarations

  setparam("XPRS_PRESOLVE", 0)
  setparam("zerotol", 5.0E-4)  ! Set Mosel comparison tolerance
  maximize(XPRS_LPSTOP, MaxProfit)  ! Solve the root LP
  savebasis(bas)  ! Save the basis

  forall(f in RF, t in 1..2) do  ! Fix some binary variables
    osol(f, t) := getsol(openm(f, t))
    if osol(f, t) = 0 then
      setub(openm(f, t), 0.0)
    elseif osol(f, t) = 1 then
      setlb(openm(f, t), 1.0)
    end-if
  end-do
maximize(XPRS_CONT,MaxProfit)       ! Solve modified problem
solval:=getobjval                    ! Save solution value

forall(f in RF, t in 1..2)          ! Reset variable bounds
    if((osol(f,t) = 0) or (osol(f,t) = 1)) then
        setlb(openm(f,t), 0.0)
        setub(openm(f,t), 1.0)
    end-if

loadbasis(bas)                      ! Load previously saved basis
setparam("XPRS_MIPABSCUTOFF", solval)  ! Set cutoff value
maximize(MaxProfit)                 ! Solve original problem
returned:= if(getprobstat=XPRS_OPT, getobjval, solval)
end-function
Mosel modules and packages
» Open architecture:
  » possibility to define language extensions via packages or modules without any need to modify the core of the Mosel language
» **Package** = library written in the Mosel language
» making parts of Mosel models re-usable
» deployment of Mosel code whilst protecting your intellectual property
» similar structure as models (keyword `model` is replaced by `package`), compiled in the same way
» included with the `uses` statement
» definition of new types, subroutines, symbols
» see examples in the *Mosel User Guide*
Module = dynamic library written in C

modules of the Mosel distribution:

- solver interfaces:
  Xpress-Optimizer (LP, MIP, QP), SLP, SP, CP
- database access: ODBC, OCI
- system commands; model handling; graphics

write your own modules for

- connecting to external software
- time-critical tasks
- defining new types, subroutines, operators, I/O drivers, control parameters, symbols
Some highlights of module features

» Interaction with external programs during their execution (callback functions)
» Access to other solvers and solving paradigms (NLP, CP)
» Implementation of graphical applications (mmive, XAD)
uses "mmxprs"

declarations
  x: array(1..10) of mpvar
end-declarations

public procedure printsol
  writeln("Solution: ", getsol(Objective))
  forall(i in 1..10) write("x(" , i , ")=" , getsol(x(i)) , ",")
  writeln
end-procedure

setcallback(XPRS_CB_INTSOL, "printsol")
» What is the greatest area of a polygon of N sides and a diameter of 1?
model "Polygon"
uses "mmxslp"

declarations
  N=5
  area: gexp
  rho, theta: array(1..N) of mpvar
  objdef: mpvar
  D: array(1..N,1..N) of genctr
end-declarations

forall(i in 1..N-1) do        ! Initialization of SLP variables
  rho(i) >= 0.1; rho(i) <= 1
  SLPDATA("IV", rho(i), 4*i*(N + 1 - i)/((N+1)^2))
  SLPDATA("IV", theta(i), M_PI*i/N)
end-do
forall(i in 1..N-2, j in i+1..N-1) ! Third side of all triangles
D(i,j) := rho(i)^2 + rho(j)^2 -
    rho(i)*rho(j)*2*cos(theta(j)-theta(i)) <= 1

! Vertices in increasing order
forall(i in 2..N-1) theta(i) >= theta(i-1) +.01
theta(N-1) <= M_PI ! Boundary conditions

area :=
    (sum(i in 2..N-1) (rho(i)*rho(i-1)*sin(theta(i)-theta(i-1))))*0.5

objdef = area; objdef is_free
SLPloadprob(objdef)
SLPmaximize

writeln("Area = ", getobjval)
end-model
Example: jobshop scheduling

- schedule the production of a set of jobs on a set of machines. Every job is produced by a sequence of tasks, each of these tasks is processed on a different machine. A machine processes at most one job at a time.

Implementation with high-level modeling objects (tasks and resources)
model "Job Shop"
uses "kalis"

declarations
  JOBS = 1..NJ ! Set of jobs
  MACH = 1..NM ! Set of resources
  RES: array(JOBS,MACH) of integer ! Resource use of tasks
  DUR: array(JOBS,MACH) of integer ! Durations of tasks

  res: array(MACH) of cpresource ! Resources
  task: array(JOBS,MACH) of cptask ! Tasks
end-declarations

... ! Initialize the data

HORIZON:= sum(j in JOBS, m in MACH) DUR(j,m)
forall(j in JOBS) getend(task(j,NM)) <= HORIZON

! Setting up the resources (capacity 1)
forall(m in MACH)
set_resource_attributes(res(m), KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations, resource used)
forall(j in JOBS, m in MACH)
set_task_attributes(task(j,m), DUR(j,m), res(RES(j,m)))

! Precedence constraints between the tasks of every job
forall (j in JOBS, m in 1..NM-1)
setsuccessors(task(j,m), {task(j,m+1)})

! Solve the problem & print solution
if cp_schedule(getmakespan)<>0 then
  writeln("Total completion time: ", getsol(getmakespan))
end-if
end-model
Module mmive:
Drawing user graphs

model "Schedule"
uses "mmive", "mmsystem"

declarations
MACHINES=6; JOBS=6
graphs, colors: array(1..MACHINES) of integer
labels: array(1..JOBS) of integer
curmachine, curjobs, n1, n2, n3: integer
end-declarations

colors:: [IVE_WHITE, IVE_YELLOW, IVE_CYAN, IVE_RED, IVE_GREEN, IVE_MAGENTA]
fopen("schedule.dat", F_INPUT)

forall (i in 1..MACHINES) do
  graphs(i):= IVEaddplot("Machine "+i, IVE_BLUE)
  labels(i):= IVEaddplot("Jobs for machine "+i, Color(i))
end-do
forall (i in 1..MACHINES) do
readln(n1, n2) ! Read machine no. & no. of jobs
writeln("Machine ", n1, " Jobs:", n2)
curmachine:= n; curjobs:= n2
forall(j in 1..curjobs) do
readln(n1, n2, n3) ! Read job no., start & finish times
writeln("On machine ", curmachine, " job ", n1,
   " starts at ", n2, " and finishes at ", n3)
IVEdrawarrow(graphs(curmachine), n2, curmachine, n3, curmachine)
IVEdrawlabel(labels(n1), (n2+n3)/2, curmachine,
   "Job "+n1+r starts: "+n2+r ends: "+n3)
end-do
end-do
IVEzoom(0, 0, 30, 7)
fclose(F_INPUT)
end-model
» Working with several models in parallel, possibly in a heterogeneous distributed architecture (module *mmjobs*)

» see whitepaper Multiple models and parallel solving with Mosel

» Combining different solvers

» see whitepaper Hybrid MIP/CP solving with Xpress-Optimizer and Xpress-Kalis
The modules of the Mosel distribution are documented in the Mosel Language Reference Manual (with separate manuals for solver modules *mmxs*/*lp* and *kalis*).

The Mosel Native Interface User Guide explains how to write your own modules.
Embedding Mosel models
Embedding models in applications
What is the Mosel API?

» The Mosel language allows you to formulate optimization problems, and develop optimization methods (i.e., use the Optimizer to solve them), as a Mosel model

» The Mosel API (also Mosel libraries) allows you to embed Mosel models in an application
Programming environments

» The Mosel API is available for C/C++, Java, .NET and VB

» We use Java in the slides, but the functionality applies to all languages, and similar applications can be developed in other languages
» Model Compiler Library
  » compiles to a virtual machine
  » binary format architecture independent

» Runtime Library
  » load and run binary (models)
  » access to Mosel internal database (data, solution values, ...)
» With Xpress-IVE: select *Deploy* ➤ *Deploy* or click the deploy button 🎉
Choose the application language:

- Save BIM file
  - With debug info
  - All names stripped
- Run Mosel model from
  - C
  - Java
  - Visual Basic
- Optimize matrix file from
  - C
  - Java
  - Visual Basic
  - VBA.net
  - C#

To directly create a Windows executable that runs a BIM file:

1. Copy C:\xpressMP\bin\mrun.exe to the same folder as the BIM file.
2. Rename mrun.exe to match the name of the BIM file, but with .EXE instead.
» Clicking on the Next button will open a new window with the resulting code
» Use the Save as button to set the name and location of the new file.
Mosel library functions

» General:

XPRM(), XPRM.getVersion, XPRM.license, ...

» Model handling:

XPRM.compile, XPRM.loadModel, XPRMModel.run, XPRMmodel.getResult, XPRMModel.getExecStatus, XPRMModel.reset, ...

» Solution information:

XPRMModel.getObjectiveValue, XPRMModel.getProblemStatus, XPRMMPVVar.getSolution, XPRMLinCtr.getActivity, ...
» Accessing model objects:

XPRMModel.findIdentifier

» Arrays:

XPRMArray.getDimension, XPRMArray.getIndexSets, XPRMArray.getFirstIndex, XPRMArray.nextIndex, XPRMArray.get, ...

» Sets:

XPRMSets.getSize, XPRMSets.getFirstIndex, XPRMSets.isFixed, ...

» Handling of modules:

XPRM.findModule, XPRM.setModulesPath, XPRMModule.parameters, ...
» Use IVE to generate a Java program that compiles and runs model `chess5.mos`.

» Modify the program so that the model execution uses the data file `chess4.dat`.

» Check the problem status and output the objective value.
import com.dashoptimization.*;

public class chessc
{
    public static void main(String[] args) throws Exception
    {
        int result;
        XPRMModel model;
        XPRM xprm;

        // Initialize Mosel
        xprm = new XPRM();

        // Load compiled model (.BIM file)
        model = xprm.loadModel("chess5.bim");
// Run model
model.execParams = "DATAFILE=chess4.dat";
model.run();
System.out.println("Model execution returned: " +
    model.getResult());

// Check problem status and retrieve the optimal solution value
if (model.getProblemStatus()==XPRMModel.PB_OPTIMAL)
    System.out.println("Objective value: " + 
        model.getObjectiveValue());

    model.reset();
}
}
Extending the example

» Retrieving detailed solution information and model data

XPRMModel model;
XPRMSet prods;
XPRMArray profit, ax;
XPRMMPVar x;
int[] idx = new int[1];
double val;

// Retrieve solution values and problem data
prods = (XPRMSet)model.findIdentifier("PRODS");
profit = (XPRMArray)model.findIdentifier("PROFIT");
ax = (XPRMArray)model.findIdentifier("x");
// Get the first entry of array 'ax'
// (we know that the array is dense and has a single dimension)
idx = ax.getFirstIndex();
do {
    x = ax.get(idx).asMPVar();       // Get a variable from 'ax'
    val = profit.getAsReal(idx);    // Get the corresponding value
    System.out.println(prods.get(idx[0]) + " : " + x.getSolution() + "\t (profit: " + val + ")");   // Print the solution value
} while(ax.nextIndex(idx));        // Get the next index
public class chessio {

static int NP = 4; // Input data
static final double[] dur = {3, 2, 2, 3};
static final double[] wood = {1, 2, 3, 6};
static final double[] profit = {5, 12, 20, 40};

// Array for solution values
static double[] solution = new double[NP];

public static void main(String[] args) throws Exception {
    int result;
    XPRMModel model;
    XPRM xprm;
Extending the example

```java
xprm = new XPRM(); // Initialize Mosel
xprm.compile("chess5ioj.mos"); // Compile + load model
model = xprm.loadModel("chess5ioj.bim");
xprm.bind("DUR", dur); // Associate Java objects with
xprm.bind("WOOD", wood); // names in Mosel
xprm.bind("PROFIT", profit);
xprm.bind("xsol", solution);
model.execParams = "NP="+NP; // Set runtime parameters
model.run(); // Run the model
if (model.getProblemStatus()==model.PB_OPTIMAL)
{
    // Check problem status and display the solution
    System.out.println("Objective: " + model.getObjectiveValue());
    for(int i=0;i<NP;i++)
        System.out.println("x(" + (i+1) + "): ": " + solution[i] + "\t(profit: " + profit[i] + ")");
}
model.reset();
```
Summary

» Mosel libraries allow you to embed model programs directly in your application
» Access the solution directly in your application, as alternative to using ODBC
» Enjoy benefits of structured modeling language and rapid deployment when building applications
May choose to work with compiled models rather than model source files – provides protection against the user viewing / changing the model.

Compiled models are platform independent.
You will find it helpful to refer to the Mosel Libraries Reference Manual.

The part ‘Working with the Mosel libraries’ of the Mosel User Guide documents examples for different programming language interfaces.
Summary and further information
Summary
» Have seen:
  » FICO Xpress product suite
» Have seen:
  » FICO Xpress product suite
  » solvers
» Have seen:
  » FICO Xpress product suite
    » solvers
    » modeling interfaces
» Have seen:
  » FICO Xpress product suite
    » solvers
    » modeling interfaces
    » development environment
Summary

» Have seen:
  » Modeling with Mosel
    » formulating Linear and Mixed Integer Programming (LP and MIP) problems
» Have seen:
  » Modeling with Mosel
    » formulating Linear and Mixed Integer Programming (LP and MIP) problems
  » accessing data sources
Summary

» Have seen:
   » Modeling with Mosel
     » formulating Linear and Mixed Integer Programming (LP and MIP) problems
   » accessing data sources
   » programming language features
» Have seen:
» Modeling with Mosel
» formulating Linear and Mixed Integer Programming (LP and MIP) problems
» accessing data sources
» programming language features
» language extensions (modules and packages)
» Have seen:
  » Modeling with Mosel
    » formulating Linear and Mixed Integer Programming (LP and MIP) problems
  » accessing data sources
  » programming language features
  » language extensions (modules and packages)
  » Embedding models in applications for deployment
Further information

» Xpress website:
  http://www.fico.com/xpress

» Examples database:
  http://examples.xpress.fico.com

» Whitepapers, documentation:
  http://optimization.fico.com