FICO™

Mosel: An Overview

FICO^{TM} Xpress Optimization Suite Whitepaper

Last update 24 June, 2008

Mosel: An Overview

Y. Colombani and S. Heipcke

Xpress Team, FICO, Leam House, Leamington Spa CV32 5YN, UK
http://www.fico.com/xpress

May 2002, last rev. June 2008

Abstract

This paper introduces the basics of the *Mosel language* that are required to use the software as a modeling and solution reporting interface to standard matrix-based solvers. Taking this a step further, it also shows how Mosel can be used to implement more complex solution algorithms.
Using the *Mosel libraries*, a model written in the Mosel language can be integrated with and accessed from application programs implemented in programming languages such as C, C++, Java, C#, Visual Basic.
The open, modular architecture of the Mosel environment for modeling and solving has been designed to be *easily extensible*, not being restricted to a particular type of problem or solver. The paper explains how the user can extend the existing Mosel language to provide new functionality that may be required, for instance, to access other solvers.

Keywords: Modeling language, programming language, multiple solvers

Contents

1 Introduction

Mosel is an environment for modeling and solving problems that is provided either in the form of libraries or as a standalone program. Mosel includes a language that is both a *modeling* and a *programming* language combining the strengths of these two concepts. As opposed to "traditional" modeling environments like AMPL [Fourer et al., 1993] for which the problem is described using a "modeling language" and algorithmic operations are written with a "scripting language" (similarly for OPL [Van Hentenryck, 1998] with OPL-script), in Mosel there is no separation between a modeling statement (*e.g.* declaring a decision variable or expressing a constraint) and a procedure that actually solves a problem (*e.g.* call to an optimizing command). Thanks to this synergy, one can program a complex solution algorithm by interlacing modeling and solving statements.

1.1 Solver Modules

Each category of problem comes with its own particular types of variables and constraints and a single kind of solver cannot be efficient in all cases. To take this into account, Mosel does not integrate any solver by default but offers a dynamic interface to external solvers provided as *modules*. Each solver module comes with its own set of procedures and functions that directly extends the vocabulary and capabilities of the language. This architecture guarantees an efficient link between Mosel and the solver(s) being used. Similar connections are also provided by other systems (*e.g.* MPL [Maximal, 2001]) but due to the concept of modules, Mosel is not restricted to any particular type of solver and each solver may provide its specifics at the user level. For instance, an LP solver may define the procedure "setcoeff(ctr,var,coeff)" to set the matrix coefficient of the constraint 'ctr' for the variable 'var' whilst such a procedure may make no sense for other types of solvers. Similarly, the module *mmquad* extends the syntax of the language with quadratic expressions that can be handled by a suitable solver (like Xpress-Optimizer).

A major advantage of the modular architecture is that there is no need to modify Mosel to provide access to a new solution technology.

Currently available solver modules for Mosel include *mmxprs* that gives access to Xpress-Optimizer for solving Linear, Mixed-Integer and Quadratic Programming problems, *mmxslp* for defining and solving problems with non-linear constraints via Sequential Linear Programming, *mmsp* for formulating and solving Stochastic Programming problems, and *kalis* for Constraint Programming.

1.2 Other modules

It may be noted here that the modular architecture of Mosel can also be used as a means to open the environment to software other than solvers. For example, one Mosel module (*mmodbc*) allows the user to *access databases and spreadsheets* that define an ODBC interface using standard SQL commands. Other libraries could be written to provide the functionalities required to communicate with a specific database (in addition to ODBC, *mmodbc* already provides a software-specific interface for Microsoft Excel).

Another Mosel module (*mmjobs*) implements facilities for handling multiple models, including mechanisms for synchronization and data exchange in memory, thus giving way to an implementation with Mosel of a large range of (decomposition) algorithms. Several examples of such algorithms are described in the Xpress Whitepaper *Multiple models and parallel solving with Mosel*.

With the help of the module *mmive* the user can create his own *graphics* in the graphical environment Xpress-IVE. Taking this even further, the module *mmxad* (Xpress Application Developer, XAD) enables the user to define a complete graphical application from within a Mosel model.

Other modules could be written to provide the functionalities required to communicate with other applications for which no generic interface is available.

1.3 User modules

Besides the provided modules, Mosel is open to any kind of addition by its users. The communication between Mosel and its modules uses specific protocols and libraries. This *Native Interface* is public and allows the user to implement his own modules. Any programming task that can be cast into the form of a C program may be turned into a module to be used from the Mosel language.

Possible uses of user-written modules include, but are not limited to,

- application specific data handling (*e.g.* definition of composite data types, data input and output in memory),
- access to external programs,
- access from the model to solution algorithms and heuristics implemented in a programming language (possibly re-using existing code),
- access to efficient implementations of specific, time-critical programming tasks (such as a sorting algorithm that needs to be called frequently by some heuristic).

1.4 Tools

The Mosel distribution contains a set of tools, including a preprocessor for models and most importantly, a *debugger* (including all standard debugging functionality like stepwise execution or conditional breakpoints) and a profiler for analyzing Mosel models. Both these tools are accessed from the Mosel command line and they are also supported by the graphical development environment Xpress-IVE. For further detail on their use the reader is refered to the "*Mosel User Guide*".

1.5 I/O drivers

The notion of "file" in Mosel has a very general meaning. A "file" may be, for instance,

- a physical file (text or binary)
- a block of memory
- a file descriptor provided by the operating system

- a function (callback)
- a database

The Mosel distribution includes a set of I/O drivers that provide interfaces to specific data sources or make it possible to exchange information between the application running the Mosel libraries and a Mosel model in a very direct way. With the help of the Mosel Native Interface (NI) the user may also implement his own drivers. Since working with I/O drivers is quite an advanced use of Mosel this functionality is explained in a separate Xpress Whitepaper entitled *Generalized file handling in Mosel*.

1.6 Contents of this paper

The first part of this paper gives an overview on the general architecture of the system and introduces its language by means of a small example. Certain aspects of the Mosel language are discussed with some more detail in the second section, accompanied by a number of program extracts. The following section explains the concept of modules in Mosel and introduces the currently commercially available modules. Subsequently, we demonstrate how Mosel can be used to implement more advanced solution algorithms. For software integration purposes, it is certainly important to know how to access from within a programming language objects that have been defined in the Mosel language: this is the topic of the next section. The last two sections describe how a user can implement his own Mosel libraries; these may take the form of packages (libraries written in the Mosel language) or new modules (C libraries using the Mosel Native Interface).

2 Overview

2.1 The Mosel language: a simple example

The following two-product (`xs` is the number of small things to make, `xl` the number of large ones), two-resource constraint production planning example shows how to write and solve an easy mixed integer programming (MIP) problem with Mosel.

General structure: Every Mosel program starts with the keyword `model`, followed by a name and terminates with `end-model`. All objects must be declared in a `declarations` section, unless they are defined unambiguously through an assignment (*e.g.* `i:=1` defines `i` as an integer and assigns to it the value 1). There may be several such `declarations` sections at different places in a model. In the present case, we define two variables `xs` and `xl` of type `mpvar`. The model then defines two linear inequality constraints on the two variables and constrains `xs` and `xl` to take only integer values.

```
model Chess
  uses "mmxprs"                       !  Use Xpress-Optimizer for solving

  declarations
    xs,xl:  mpvar                     !  Decision variables
  end-declarations

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

  xs is_integer; xl is_integer        !  Integrality constraints

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

  writeln("Solution:  ", getobjval)   !  Print objective function value
  writeln("small:  ", getsol(xs))     !  Print solution values for variables xs
  writeln("large:  ", getsol(xl))     !  and xl
end-model
```

Solving: With the procedure `maximize`, we call Xpress-Optimizer to maximize the linear expression `5*xs + 20*xl`. Since there is no "default" solver in Mosel, we specify that Xpress-Optimizer is to be used with the statement `uses "mmxprs"` at the begin of the program.

Output printing: The last three lines print out the value of the optimal solution and the solution values for the two variables. Instead of writing `getsol(xs)` we could equally have used the dot notation style, `xs.sol`.

Line breaks: Note that it is possible to place several statements on a single line, separating them by semicolons (like `xs is_integer; xl is_integer`). On the contrary, since there are no special "line end" or continuation characters, every line of a statement that continues over several lines must end with an operator (+, >= etc.) or characters like , that make it obvious that the statement is not terminated.

Comments: As shown in the example, single line comments in Mosel are preceded by `!`. Comments over multiple lines start with `(!` and terminate with `!)`.

2.2 Extending the example

To demonstrate the expressive power of Mosel, below follows an enhanced version of the previous model.

Naming constraints: In this example, constraints are named – we no longer use a linear expression but simply its reference when we call the solver; the references to the two inequalities could, for instance, be used to access solution information (dual, slack, activity) for these constraints.

Data structures: This second example also introduces the data structures `set` and `array`. Here both array `DescrV` and its indexing set `Allvars` are dynamic since their size and contents are not known at their creation. Their contents is defined by the assignments that succeed the `declarations` section.

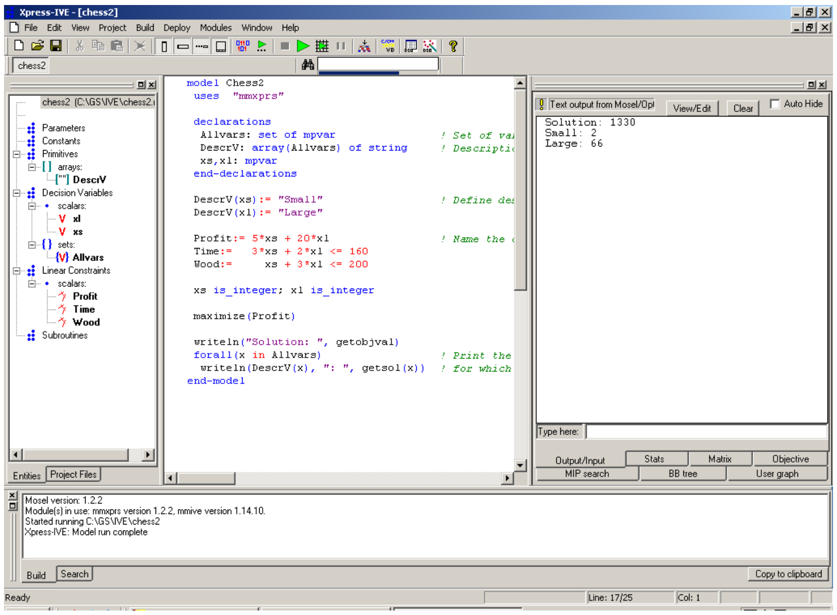
```
model Chess2
  uses "mmxprs"

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

  DescrV(xs):= "Small"                      !  Define descriptions for variables
  DescrV(xl):= "Large"

  Profit:= 5*xs + 20*xl                     !  Name the objective function
  Time:= 3*xs + 2*xl <= 160
  Wood:= xs + 3*xl <= 200

  xs is_integer; xl is_integer

  maximize(Profit)

  writeln("Solution:  ", getobjval)
  forall(x in Allvars)                      !  Print the solutions of all variables
    writeln(DescrV(x), ":  ", getsol(x))    !  for which a description is defined
end-model
```

Figure 1 shows the result of an execution of our model in the development environment Xpress-IVE.

3 The language

3.1 Types and data structures

Mosel provides the *basic types* that may be expected of any programming language: `integer`, `real` (double precision floating point numbers), `boolean` (symbols `true` and `false`), `string`



```
model Chess2
 uses  "mmxprs"

 declarations
  Allvars: set of mpvar                   ! Set of va
  DescrV: array(Allvars) of string        ! Descripti
  xs,xl: mpvar
 end-declarations

 DescrV(xs):= "Small"                     ! Define des
 DescrV(xl):= "Large"

 Profit:= 5*xs + 20*xl                    ! Name the
 Time:=   3*xs + 2*xl <= 160
 Wood:=     xs + 3*xl <= 200

 xs is_integer; xl is_integer

 maximize(Profit)

 writeln("Solution: ", getobjval)
 forall(x in Allvars)                     ! Print the
  writeln(DescrV(x), ": ", getsol(x))     ! for which
end-model
```

```
Solution: 1330
Small: 2
Large: 66
```

```
Mosel version: 1.2.2
Module(s) in use: mmxprs version 1.2.2, mmive version 1.14.10.
Started running C:\GS\IVE\chess2
Xpress-IVE: Model run complete
```

Figure 1: IVE display after model execution

(single character and any text). Together with the *MP types* `mpvar` (decision variables) and `linctr` (linear constraints) that are provided specifically for mathematical programming, these form the *elementary types* of Mosel.

In addition to the elementary types, Mosel defines the structured types `set` (collection of elements of a given type), `array` (collection of labeled objects of a given type), `list` (ordered collection of elements of a given type), and `record` (collection of objects of any type). As mentioned in the previous section, the data structures (sets and arrays) in Mosel are *dynamic* if at their creation their size and/or contents is not known. The following definitions result in *constant* sets (`R1`, `S1`) and list s(`L1`) and *static* arrays (`A1`, `A2`):

```
declarations
  R1 = 3..5
  S1 = {"red", "green", "blue"}
  A1:  array(S1) of real
  A2:  array(R1, -2..1) of mpvar
  L1 = [1,2,3,4,5]
end-declarations
```

Dynamic sets (`R2`, `S2`), lists (`L2`, `L3`) and arrays (`A3`, `A4`) are created with definitions like

```
declarations
  S2:  set of string
  A3:  array(S2,S2) of linctr
  A4:  array(R2:range) of boolean
  L2:  list of real
  L3:  array(set of integer) of list of string
end-declarations
```

The examples introduce a special type of set: `R1` and `R2` are *ranges* (= ordered sets of integers). `3..5` indicates that `R1` is the set of all integers from 3 to 5.

The definition of a record has some similarities with the `declarations` block: it starts with the keyword `record`, followed by a list of field names and types, and the keyword `end-record` marks the end of the definition. The definition of records must be placed in a `declarations` block. The following code extract defines a record with three fields ('Source', 'Sink', and 'Cost') that may be used to represent an arc in a network.

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

OneArc.Source:= "A"; OneArc.Sink:= "B"
ARCS(1).Cost:= 5.1
```

The user may define new types that will be treated in the same way as the predefined types of the Mosel language. The naming of the record 'arc' above is an example of such a *user type definition* for records.

3.2 Initialization of data/data file access

Data structures may be assigned values directly in the Mosel model or they are initialized with values read in from an external file. Here are some examples of value assignments in a Mosel model.

```
declarations
  V: real
  A1:  array(3..5) of real
  S: set of string
  L: list of integer
  A2:  array(range,range) of boolean
  AL: array(set of string) of list of real
end-declarations

V := 0.5                                  !  Assign a value to a scalar
A1 ::  [1.5, 2.3, 4.5]                    !  Initialize an array with known index range
A1(3)  := 1.8                             !  (Re)Assign a single array entry

S := {"A", "BC", "DEF"}                   !  Assign a set
L := [1, 2, 3, 3, 4, 5, 4]                !  Assign a list

A2 ::  (1..2 0..2)[true, false, false,    !  Initialize a 2-dim.  array
              true, true, false]

AL ::  (["A", "B"])[[3, -6, 1.5], [2, 8.4]]  !  Initialize an array of lists
AL("C")  := [1, 2.5, 4, -0.5]             !  Assign an array entry
```

When data are read from a file most often dynamic data structures are used, as is the case in this example extract that illustrates the use of the `initializations` section in Mosel.

```
declarations
  A: array(1..6) of real                     !  Static array definition
  S: set of string                           !  Dynamic set
  C: array(S) of real                        !  Dynamic array
  L: array(set of integer) of list of string !  Dynamic array of lists
  R: array(range) of arc                     !  Dynamic array of records
end-declarations

initializations from "initdata.dat"
  A C S L R
end-initializations

writeln("I = ", I, "\nA = ", A, "\nC = ", C )
writeln("S = ", S, "\nL = ", L, "\nR = ", R)
```

The data file `initdata.dat` that is read by this program has the following contents:

```
I: 10
A: [2 4 6 8]
C: [("red") 3 ("green") 4.5 ("blue") 6 ("yellow") 2.1]
S: ["white" "black"]
L: [(1) ["a" "b"] (3) ["c" "d" "e" "f"] (6) ["i" "i" "i"]]
R: [(1) ["A" "B" 1.2] (2) ["A" "C" 4.5] (3) ["B" "C" 3] ]
```

For the static array A the indices are known, the values may therefore be given as a simple list. For dynamic arrays, the data file also needs to contain the indices.
With this data file, the program produces the following output:

```
I = 10
A = [2,4,6,8,0,0]
C = [(‘red’,3),(‘green’,4.5),(‘blue’,6),(‘yellow’,2.1)]
S = {‘red’,‘green’,‘blue’,‘yellow’,‘white’,‘black’}
L = [(1,[‘a’,‘b’]),(3,[‘c’,‘d’,‘e’,‘f’]),(6,[‘i’,‘i’,‘i’])]
R = [(1,[Source=‘A’ Sink=‘B’ Cost=1.2]),(2,[Source=‘A’ Sink=‘C’ Cost=4.5]),
(3,[Source=‘B’ Sink=‘C’ Cost=3])]
```

In the static array A all entries are defined, hence there is no need to specify the indices. For dynamic arrays a list of n-tuples is printed where the first n-1 elements are the indices and the last the value of the array entry.

For more flexibility, it is also possible to use the procedures read/readln, or write/ writeln to read from or write to text files. To read the data file readdata.dat with the following contents

```
# Data:
A(1) = 5.2
A(2) = 3.4
```

we may write the following code in a Mosel program:

```
declarations
  j:  integer
  B: array(range) of real
end-declarations

fopen("readdata.dat", F_INPUT)

fskipline("#")                    !  Skip lines starting with a '#'
repeat
  readln('A(',j,') =',B(j))       !  Read the indices and the value on a line
until (getparam("nbread")<4)      !  until a line is not properly constructed
fclose(F_INPUT)
```

3.3 Language constructs

Besides data types, Mosel also provides the typical flow controls (selections and loops) that one will expect from a programming language.

3.3.1 Selections

The simplest form of a selection is the if-then statement which may be extended to if-then-else or even if-then-elif-then-else to test two conditions consecutively and execute an alternative if both fail as in the following example:

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

The upper bound 7 is applied to the variable *x* if the value of *A* is greater or equal 20, and if the value of *A* is less or equal 10 then the lower bound 35 is applied to *x*. In all other cases (that is, *A* is greater than 10 and smaller than 20), *x* is fixed to 0.

If several mutually exclusive conditions (here: values of *A*) are tested, the `case` statement should preferably be used as in

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

3.4 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`). The `forall` and `while` loops in Mosel exist in two versions: an inline version for looping over a single statement as in

```
declarations
  x:  array(1..10) of mpvar
end-declarations
forall(i in 1..10) x(i) is_binary
```

and a second version `forall-do` (`while-do`), that may enclose a block of statements, the end of which is marked by `end-do`:

```
declarations
  x:  array(1..10) of mpvar
end-declarations
forall(i in 1..10) do
  x(i) is_integer
  x(i) <= 100
end-do
```

3.5 Example: working with sets

The following example introduces operations on sets and demonstrates the use of different types of (nested) loops. This program calculates the set of prime numbers between 2 and some given upper limit using the “Sieve of Eratosthenes” (for every prime number that is found all of its multiples are deleted from the set of remaining 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, ":")
  n:=2
  repeat
    while (not(n in SNumbers)) n+=1
    SPrime += {n}
    i:=n
    while (i<=LIMIT) do
      SNumbers-= {i}
      i+=n
    end-do
  until SNumbers={}

  writeln(SPrime)
end-model
```

Set operators: Subsets may be added or removed from a set using the operators += and -=. (Note that a set may not decrease in size once it is used as an indexing set.) Mosel also defines the standard operations on sets: union, intersection and difference (operators +, *, -).

Run-time parameters: This example introduces the `parameters` section: the value of constants defined in this section may be reset at the execution of the model, otherwise their given default value is used. Here we enable the person who runs the program to choose the upper limit of the set of numbers; another typical use may be to specify the name of data file(s) in the form of parameters.

3.6 Subroutines

Mosel provides a set of predefined subroutines (*e.g.* procedures like `write` / `writeln`, arithmetical functions like `cos`, `exp`, `ln`, or subroutines to access certain model objects such as `getsol`, `reverse`, `getsize`), but it is also possible to define new procedures and functions according to the needs of a specific program. 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. It is possible to pass parameters into a subroutine. The (list of) parameter(s) is added in parentheses following the name of the subroutine.

```
model "Simple subroutines"
  function timestwo(b:integer):integer
    returned := 2*b
  end-function

  procedure printstart
    writeln("The program starts here.")
  end-procedure

  printstart
  a:=3
  writeln("a = ", a)
  a:=timestwo(a)
  writeln("a = ", a)
end-model
```

The structure of subroutines is very similar to the one of `model`: they may include `declarations` sections for declaring local parameters that are only valid in the corresponding subroutine. Subroutine calls may be nested, and they may also be called recursively.

Forward declaration: Mosel enables the user to declare a subroutine separately from its definition by using the keyword `forward`.

Overloading: 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*. The user may define (additional) overloaded versions of any subroutines defined by Mosel and also for his own functions and procedures.

4 Mosel libraries

Models written with the Mosel language can be accessed from C through the Mosel C interface (other interfaces available: Java, C#, Visual Basic). This interface is provided in the form of two libraries; it may especially be of interest for integrating models and/or solution algorithms written with Mosel into some larger system, (re)using already existing parts of algorithms written in C, and for interfacing Mosel with other software.
The Mosel Model Compiler Library needs to be used to compile a model file into a BIM file (portable **Bi**nary **M**odel file). This BIM file is then input with the Mosel Run Time Library for executing the model.

Using the Mosel libraries, it is not only possible to compile and run models, but also to access information on the different modeling objects. The following example shows how to compile the model `Prime` presented in Section 3.5, execute it with a different value for the parameter `LIMIT`, and print the resulting set of prime numbers. (The example actually works with a model `Prime2` that contains no output printing because this is done in C).
To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), one first needs to retrieve the Mosel reference to this object using function `XPRMfindident`. It is then possible to enumerate the elements of the set and obtain their respective values.

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

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

  XPRMinit();
  XPRMcompmod(NULL, "prime2.mos", NULL, NULL);  /* Compile model Prime2 */
  mod=XPRMloadmod("prime2.bim", NULL);          /* Load the BIM file */
  XPRMrunmod(mod, &result, "LIMIT=500");        /* Run the model */

  XPRMfindident(mod, "SPrime", &rvalue);        /* Get the object 'SPrime' */
  set = rvalue.set;
  size = XPRMgetsetsize(set);                   /* Get the size of the set */
  if(size>0)
  {
    first = XPRMgetfirstsetndx(set);            /* Get number of the first index */
    last = XPRMgetlastsetndx(set);              /* Get number of the last index */
   printf("Prime numbers from 2 to 500:\n");
    for(i=first;i<=last;i++)                    /* Print all set elements */
      printf(" %d,", XPRMgetelsetval(set,i,&setitem)->integer);
    printf("\n");
  }

  XPRMfinish();
  return 0;
}
```

5 Modules

An original feature of Mosel is the possibility to extend the language by means of *modules* (dynamic libraries written in the C programming language that observe the conventions set out by the Mosel Native Interface). A module may extend the Mosel language with new

– constant symbols
– subroutines
– types
– I/O drivers
– control parameters

In this list, subroutines and types are certainly the most important items. *Subroutines* defined by a module may be entirely new functions or procedures or overload existing subroutines of Mosel. A module may, for instance, provide a subroutine that calls an algorithm or a solution heuristic that is readily available in the form of a C or C++ library function.

New *types* defined by a module are treated exactly like the own types of Mosel (like `integer` or `mpvar`). They can be used in complex data structures (arrays, sets *etc.*), read in from file in `initializations` sections, or appear as parameters of subroutines. The operators in Mosel can be overloaded to work with new types. The definition of new types may be required to support solvers such as finite domain constraint solvers.

The Mosel distribution comes with a set of *I/O drivers* that provide interfaces to specific data sources (such as ODBC) or serve to exchange information between the application running the Mosel libraries and a Mosel model in a very direct way by providing various possibilities of passing data back and forth in memory. The user may define additional drivers, for instance to read/write compressed or encrypted files.

Constants and *control parameters* published by a module make little sense on their own. They will typically be used in conjunction with its types or subroutines.

Modules may be seen as an appropriate means for rapid prototyping of solution algorithms that involve a combination of solution strategies or solvers originating from different areas of research. For example, using *mmxprs* and a module that provides access to a finite domain constraint solver, such as *kalis*, it is possible to define a MIP search that at every node in the branching tree solves subproblem(s) using constraint propagation algorithms and depending on the results, generates cuts for the MIP problem (for example implementations please see the Xpress Whitepaper *Hybrid MIP/CP solving with Xpress-Optimizer and Xpress-Kalis*).

5.1 Available modules

At present, the following modules have been implemented:

Solvers: mmxprs, mmquad, mmnl, mmxslp, mmsp, kalis

Data handling: mmodbc, mmoci, mmetc

System: mmsystem

Model handling: mmjobs

Graphics: mmive, mmxad

In the preceding examples the module *mmxprs* has already been used to solve problems with Xpress-Optimizer. Besides making the basic solution tasks and algorithm settings accessible from the Mosel language, an interesting feature of this module is the possibility to define the *callback functions* of the underlying solver C library from within Mosel as is shown in the following

program extract.
The following example defines a function for printing out the current solution that is called whenever an integer solution is found.

```
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)), "\t")
  writeln
end-procedure

setcallback(XPRS_CB_INTSOL, "printsol")
```

With the help of the module *mmquad* it is possible to formulate and solve Quadratic Programming problems (see the example in Section 5.3).

A recent addition is the module *mmnl* for handling non-linear constraints. This module is not a solver on its own. In combination with *mmxprs* you can formulate and solve Quadratically Constrained Quadratic Programming problems and Linearly Constrained Convex Optimization problems.

General non-linear problems can be formulated and solved with the Successive Linear Programming module *mmxslp*.

The module *mmsp* provides support for Stochastic Programming within Mosel.

The module *kalis* gives access to the Constraint Programming solver Kalis by Artelys.

Modules may define additional interfaces to data files:

mmetc defines the procedure `diskdata` that emulates the data in- and output of mp-model[Dash, 1999];

mmoci defines a specific interface to Oracle databases;

mmodbc provides access to any data source for which an ODBC interface is available, using Mosel's `initializations` blocks or, for larger flexibility, through standard SQL commands. This module also defines a software-specific interface to Excel.

The following program extract reads a 2-dimensional array and its sizes from an MS-Excel spreadsheet through an ODBC connection. Notice that switching to a different data source, such as a database, simply amounts to changing the file name:

```
model sizes
  uses "mmodbc"

  declarations
    Nprod, Nrm:  integer
  end-declarations

  initializations from 'mmodbc.odbc:ssxmpl.xls'
    Nprod Nrm
  end-initializations

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

  initializations from 'mmodbc.odbc:ssxmpl.xls'
    PneedsR as 'USAGE'
  end-initializations
end-model
```

The module *mmsystem* provides functions like `gettime` and file handling facilities. It even makes it possible to use operating system commands from within the Mosel language – the latter quite obviously at the expense of the portability of the model file.

The module *mmive* allows users working with Mosel models through the graphical user interface Xpress-IVE to draw their own graphs, such as the chart in Figure 2.

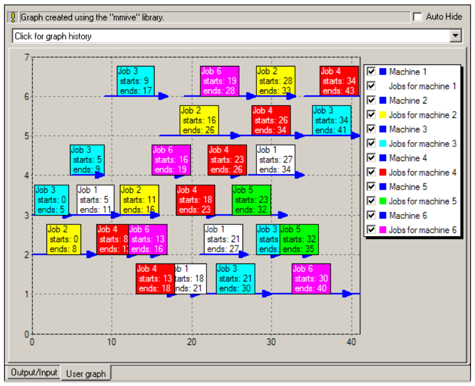
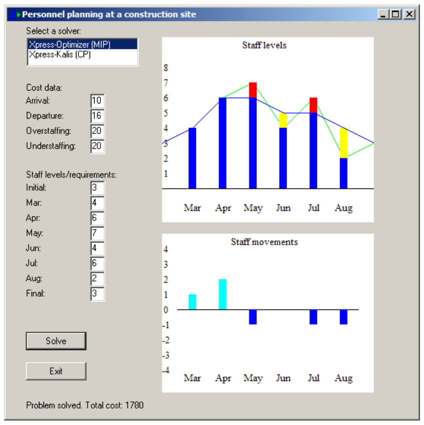


Figure 2: Gantt chart drawn by mmive

The Xpress Application Designer (module *mmxad*) lets you create complete graphical applications with Mosel such as the personnel planning application shown in Figure 3.


Personnel planning at a construction site

Select a solver:
Xpress-Optimizer (MIP)
Xpress-Kalis (CP)

Cost data:

Arrival:	10
Departure:	16
Overstaffing:	20
Understaffing:	20

Staff levels/requirements:

Initial:	3
Mar:	4
Apr:	6
May:	7
Jun:	4
Jul:	6
Aug:	2
Final:	3

Problem solved. Total cost: 1780

Figure 3: XAD application: planning the personnel at a construction site

5.2 *mmxprs*: variable fixing heuristic for MIP

In this section we give an example of a solution heuristic for solving a mixed integer programming (MIP) problem using Xpress-Optimizer (module *mmxprs*). All subroutines, types, constants and control parameters that are contributed to the Mosel language by this solver module are highlighted in bold face.

To aid structuring the implementation of this problem, the problem formulation and the solution algorithm are not only split into several subroutines, but also contained in different files that are included by the main model file:

```
model "Fixing binary variables"
  uses "mmxprs"

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

  solution:=solve

  writeln("The objective value is:  ", solution)
end-model
```

The following is an extract of the model definition, contained in file `fixbv_pb.mos`:

```
declarations
  RF=1..2 !  Range of factories (f)
  RT=1..4 !  Range of time periods (t)

  (...)

  open:  array(RF,RT) of mpvar
end-declarations

(...)

forall(f in RF,t in RT) open(f,t) is_binary
```

The model contains binary variables `open` for which a variable fixing heuristic consisting of the following steps may be implemented:

- Solve the LP problem.
- Fix the binary variables that are almost 0 or 1 at these values ("rounding").
- Solve the resulting MIP problem and retrieve the solution value.
- Restore the original MIP problem and solve it using the solution value of the modified problem as bound ("cutoff" value).

The function `solve` that implements this solution heuristic is defined in the file `fixbv_solv.mos`:

```
function solve:real
  declarations
    TOL=5.0E-4
    osol:  array(RF,RT) of real
    bas:  basis
  end-declarations

  setparam("zerotol", TOL)                           !  Set Mosel comparison tolerance
  setparam("XPRS_CUTSTRATEGY", 0)
  setparam("XPRS_HEURSTRATEGY", 0)
  setparam("XPRS_PRESOLVE", 0)

  maximize(XPRS_TOP, MaxProfit)                      !  Solve the LP problem
  savebasis(bas)                                     !  Save the current basis

  forall(f in RF, t in RT) do                        !  "Round" binaries
    osol(f,t):= getsol(open(f,t))
    if osol(f,t) = 0 then
      setub(open(f,t), 0)
    elif osol(f,t) = 1 then
      setlb(open(f,t), 1)
    end-if
  end-do

  maximize(MaxProfit)                                !  Solve the modified MIP
  ifgsol:=false
  if getprobstat=XPRS_OPT then                       !  If an integer feas.  solution was found
    ifgsol:=true
    solval:=getobjval                                !  Get the value of the best solution
  end-if

  forall(f in RF, t in RT)                           !  Restore the original problem
   if ((osol(f,t) = 0) or (osol(f,t) = 1)) then
      setlb(open(f,t), 0); setub(open(f,t), 1)
    end-if

  loadbasis(bas)                                     !  Reload the basis
  if ifgsol then                                     !  Set the "cutoff" to the
    setparam("XPRS_MIPABSCUTOFF", solval)            !  best known solution
  end-if
  maximize(MaxProfit)                                !  Solve the original MIP
  returned:=if(getprobstat=XPRS_OPT,getobjval,solval)
end-function
```

5.3 *mmquad*: defining and solving a QP

As mentioned earlier, the module *mmquad* can be used to formulate and solve Quadratic Programming (QP) problems. This module defines a new type `qexp`, that is accepted as input by the Xpress QP solver. This means, the extension to the Mosel language provided by the module *mmquad* can even be used by other modules (*inter-module communication*).

In the following example we wish to determine the composition of a portfolio that minimizes the total cost, subject to upper bounds on the assets and a limit on the total number of values that may be chosen. The dependencies between the assets under consideration lead to a quadratic cost function.

```
model Portfolio
  uses "mmxprs","mmquad"

  parameters
    DATAFILE = "portf.dat"                ! Name of the data file
    LIMIT = 20                            ! Maximum number to be chosen
  end-parameters

  declarations
    NVAL = 30                             ! Total number of assets
    RV = 1..NVAL
    LCOST: array(RV) of real              ! Coeff. of linear part of the obj.
    QCOST: array(RV,RV) of real           ! Coeff. of quadratic part of the obj.
    UBND: array(RV) of real               ! Upper bound values
    n:  integer                           ! Counter for chosen assets

    x:  array(RV) of mpvar                ! Amount taken into the portfolio
    y:  array(RV) of mpvar                ! 1 if asset i is chosen, else 0
    Cost:  qexp                           ! Objective function
  end-declarations

  initializations from DATAFILE
    UBND LCOST QCOST
  end-initializations

  Cost:= sum(i in RV) ( LCOST(i)*x(i) +     ! Define the (quadratic) cost function
         QCOST(i,i)*x(i)^2 +
        sum(j in i+1..NVAL) QCOST(i,j)*x(i)*x(j) )

  sum(i in RV) x(i) = 100                 ! Amounts chosen must add up to 100%
  sum(i in RV) y(i) <= LIMIT              ! Limit on total number of values

  forall(i in RV) do
    x(i) <= UBND(i)*y(i)                  ! Upper limits
    y(i) is_binary                        ! Variables are binary
  end-do

  minimize(Cost)                          ! Minimize the total cost

  writeln("Solution:  ", getobjval)       ! Solution printing
  writeln("(quadratic part:  ",
          getsol(sum(i in RV)  ( QCOST(i,i)*x(i)^2 +
          sum(j in i+1..NVAL) QCOST(i,j)*x(i)*x(j))), ")" )

  forall(i in RV)
    if(getsol(y(i)) > 0.000001) then
      writeln(i, ":  ", getsol(x(i)))
      n+=1
    end-if
 writeln("\n", n, " assets have been selected")
end-model
```

6 Packages

The Mosel language is open to extensions through user-written libraries. These libraries may take two forms:

- *Package* – a library written in the Mosel language defining new constants, subroutines and types for the Mosel language.
- *Module* – a dynamic library (Dynamic Shared Object, DSO) written in the C programming language.

In the remainder of this section we shall discuss an example of a user package. Two examples of user modules are described in the next section.

The structure of packages is similar to models, replacing the keyword `model` by `package`. Packages are included into models with the `uses` statement, in the same way as this is the case for modules. Unlike Mosel code that is included into a model with the `include` statement, packages are compiled separately, that is, their contents are not visible to the user.

Typical uses of packages include

- development of your personal 'tool box'

- model parts (*e.g.*, reformulations) or algorithms written in Mosel that you wish to distribute without disclosing their contents
- add-ons to modules that are more easily written in the Mosel language

6.1 Defining a new subroutine

After solving a problem, users often wish to retrieve the solution and store it. Mosel provides the function `getsol` to access solution values one-by-one, but there is no subroutine for accessing and saving the solution to an array of decision variables all at once. However, such a subroutine can easily be implemented in the form of a module. In the example from the previous section, we would then be able to print out the solution as follows:

```
model Portfolio
  uses "solarray", "mmxprs"
  ...                           !  Data initialization

  declarations
    x:  array(RV) of mpvar      !  Amount taken into the portfolio
    sol:  array(RV) of real     !  Solution values
  end-declarations

  ...                           !  Formulate and solve the problem

  solarray(x,sol)               !  Retrieve the solution for all variables
  writeln(sol)                  !  Print the solution
end-model
```

The following Mosel code produces a package ’solarray’ that may be used in the place of the module ’solarray’ we have discussed earlier. Whereas the module only defines a single C function that deals with any type and number of index sets to the arrays, we here need to define explicitly one overloaded version of the subroutine `solarray` for every configuration of array index sets we may wish to use.

```
package solarraypkg
  public procedure solarray(x:array(R:set of integer) of mpvar,
             s:array(set of integer) of real)
    forall(i in R) s(i):=getsol(x(i))
  end-procedure

  public procedure solarray(x:array(R1:set of integer,
             R2:set of integer) of mpvar,
             s:array(set of integer,
             set of integer) of real)
    forall(i in R1, j in R2) s(i,j):=getsol(x(i,j))
  end-procedure

  public procedure solarray(x:array(R1:set of integer,
             R2:set of integer,
             R3:set of integer) of mpvar,
             s:array(set of integer,
             set of integer,
             set of integer) of real)
    forall(i in R1, j in R2, k in R3) s(i,j,k):=getsol(x(i,j,k))
  end-procedure
end-package
```

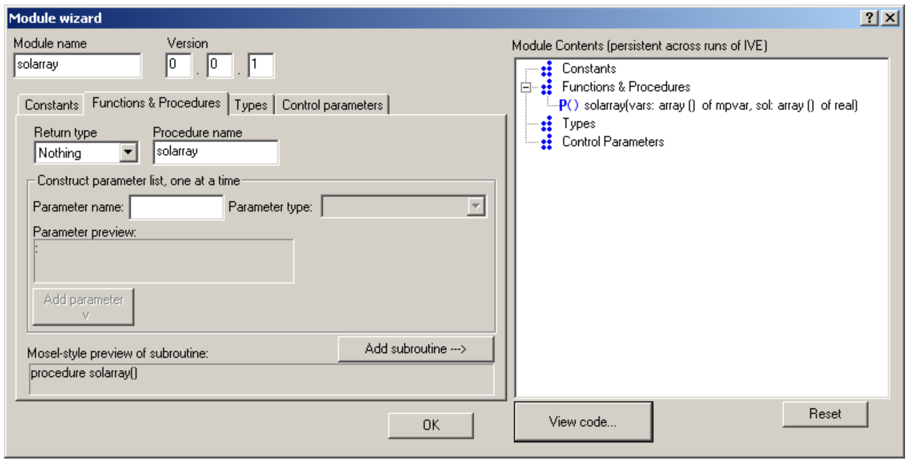
This package code, saved as `solarray.mos`, is compiled in the same way as any standard Mosel model to a BIM file (`solarray.bim` where the filenamne indicates the name of the package). If it is not included in the working directory, the environment variable `MOSEL_DSO` needs to be set to its location.

7 Writing user modules

From the operating system point of view, a module is a dynamic library (Dynamic Shared Object, DSO) written in the C programming language. The Mosel Native Interface is a set of conventions that a DSO must respect to be accepted as a module by Mosel. Any functionality that can be implemented in the form of a C library may be made accessible from within the Mosel language. In this section, we show how to define (1) a new subroutine and (2) a new type with operators to work with it.

7.1 Defining a new subroutine

The `solarray` function of package 'solarray' described in the previous section may equally be defined by a module 'solarray' – without any need for modifications to the Mosel model using this functionality. The 'solarray' module defines a single C function that deals with any type and number of index sets to the arrays, whereas with the Mosel language we need to define explicitly one overloaded version of the subroutine `solarray` for every configuration of array index sets we may wish to use.



Module name: solarray
Version: 0 . 0 . 1
Return type: Nothing
Procedure name: solarray
Mosel-style preview of subroutine: procedure solarray()

Module Contents (persistent across runs of IVE)
- Constants
- Functions & Procedures
  - P() solarray(vars: array () of mpvar, sol: array () of real)
- Types
- Control Parameters

Figure 4: Module definition with IVE

The code for implementing a module that provides this new function is shown in the following. The various interface structures, the initialization function, and also the prototype of the library function `ar_getsol` implementing "solarray" can be generated automatically with the module generation functionality in Xpress-IVE (Figure 4). The only work left over to the user is to fill in the body of the function `ar_getsol`. For simplicity's sake we have left out the error handling that should be performed by this function, we merely show the required operations:

1. Get the references to the two arrays (one array of decision variables and one array of reals) from the Mosel stack.
2. Enumerate all defined entries in the array of variables (it may be sparse or dense, with up to `MAXDIM` dimensions).
3. Get the solution value for each entry and copy it into the array of reals.

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

 #define MAXDIM 20

static int ar_getsol(XPRMcontext ctx,void *libctx);

 /* List of subroutines */
 static XPRMdsofct tabfct[]=
 {
   {"solarray", 1000, XPRM_TYP_NOT, 2, "A.vA.r", ar_getsol}
 };

 /* Interface structure */
 static XPRMdsointer dsointer=
 {
    0, NULL,
   sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
    0, NULL,
    0, NULL
 };

/* Structure for getting function list from Mosel */
 static XPRMnifct mm;

/* Module initialization function */
DSO_INIT solarray_init(XPRMnifct nifct, int *interver, int *libver,
                       XPRMdsointer **interf)
 {
    mm=nifct;                          /* Get the list of Mosel functions */
    *interver=XPRM_NIVERS;             /* Mosel NI version */
    *libver=XPRM_MKVER(0,0,1);         /* Module version:  must be <= Mosel NI version */
    *interf=&dsointer;                 /* Pass info about module contents to Mosel */

    return 0;
 }


static int ar_getsol(XPRMcontext ctx,void *libctx)
 {
    XPRMarray varr, solarr;
    XPRMmpvar var;
    int indices[MAXDIM];

/* Get variable and solution arrays from stack in the order that they are
   used as parameters for 'getsol' */
    varr=XPRM_POP_REF(ctx);
    solarr=XPRM_POP_REF(ctx);

 /* Error handling:
   - compare the number of array dimensions and the index sets
   - make sure the arrays do not exceed the maximum number of dimensions MAXDIM
 */

/* Get the solution values for all variables and copy them into the solution
    array */
   if(!mm->getfirstarrtruentry(varr,indices))
       do
       {
         mm->getarrval(varr,indices,&var);
         mm->setarrvalreal(ctx,solarr,indices,mm->getvsol(ctx,var));
      } while(!mm->getnextarrtruentry(varr,indices));

    return XPRM_RT_OK;
 }
```

This code needs to be compiled into a dynamic library giving it the extension `.dso`. Afterwards its location must be made known to Mosel (by setting the environment variable `MOSEL_DSO`) and then it may be used like any module of the Mosel distribution.

As can be seen from this example, there may be a choice whether to implement language extensions in the form of a module or as a package. Packages behave like standard Mosel models, modules are written in a lower level language, implying in general a higher development effort but also a faster execution speed. Certain functionality (such as arbitrary array indices in this example or the definition of operators in the complex number example below) can only be provided by modules.

7.2 Creating a new type

In this section we show how to implement a new type *complex* to represent complex numbers so as to make possible writing models like the following with Mosel:

```
model Complex numbers
  uses "complex"

  declarations
    c:complex                              !  Define a single complex number
    t:array(1..10) of complex              !  Define an array of complex numbers
  end-declarations

  forall(j in 1..10)
    t(j):=complex(j,10-j)                  !  Initialize with 2 integers or reals
  t(5):=complex("5+5i")                    !  Initialize with a string

  c:=prod(i in 1..5) t(i)                  !  Aggregate PROD operator
  if c<>0 then                             !  Comparison with an integer or real
    writeln("Product:  ", c)               !  Printing a complex number
  end-if

  writeln("Sum:  ", sum(i in 1..10) t(i))  !  Aggregate SUM operator
                                           !  Arithmetic operators
 c:=t(1)*t(3)/t(4) + if(t(2)=0, t(10), t(8)) + t(5) - t(9)

  initializations to "complex_out.dat"     !  Output to a file
    c t
  end-initializations
end-model
```

Besides some standardized initialization and type creation functions, the module implementing this new type defines constructors, basic arithmetic operations, the equality comparison operator and a printing function.

Once a type is defined, it can automatically be used in Mosel data structures (set, array) as shown in the example. From given defitions of the basic arithmetic operations Mosel deduces the definition of aggregate operators like aggregate products or sums, and where applicable, commutations of the operands or negations. Similarly, the definition of an equality comparison suffices to derive the inequality and from basic logical operators (none defined in this case) the definition of aggregate operators is generated. With the definition of a printing function, the output to a file with `initializations to` is also available.

Since the complete code of this module[1] occupies several pages of text, we shall restrict ourselves here to highlighting some key features of this module:

- module context
- type creation and deletion
- type transformation to and from string
- overloading of arithmetic operators

As with the previous example, we assume that the module code generation facility of IVE has been used to create the required interface structures so that we only need to fill in the corresponding function bodies.

7.2.1 Module context

A module needs to keep track of all objects created during the execution of a model so that all allocated space may be freed when the execution is terminated. This function is fulfilled by the module *context*. In this example, the context may be nothing but a chained list of complex numbers:

[1]Complete source code available from the authors.

```
typedef struct
{
  s_complex *firstcomplex;
} s_cxctx;
```

which assumes that a complex number is represented by the following structure:

```
typedef struct Complex
{
  double re, im;
  int refcnt;
  struct Complex *next;
} s_complex;
```

A module context can also be used to store the current values of control parameters or any other information that needs to be preserved between different calls to the module functions during the execution of a model.

A reset *service function* is called at the beginning and the termination of the execution of a Mosel program that uses the module. At its first call, the reset function creates and initializes a context for the model, and deletes this context (and any other resources used by the module for this model) at the second call.

7.2.2 Type creation and deletion

The objective of the type instance creation and deletion functions is to handle (create/initialize or delete/reset) the C structures that represent the external type and to update correspondingly the information stored in the module context. In this example we implement just a rudimentary memory management for the objects (complex numbers) created by the module: every time a number is created, we allocate the corresponding space and deallocate it when it is deleted. More realistically, a module may allocate chunks of memory and recycle space that has been allocated earlier by this module.

We define the *creation function* for a complex number as follows—if the number already exists we increase its *reference counter* otherwise we allocate and initialize a new C structure for this number:

```
static void *cx_create(XPRMcontext ctx, void *libctx, void *todup, int typnum)
{
  s_cxctx *cxctx;
  s_complex *complex;

  if(todup!=NULL)
  {
    ((s_complex *)todup)->refcnt++;
    return todup;
  }
  else
  {
    cxctx=libctx;
   complex=(s_complex *)malloc(sizeof(s_complex));
    complex->next=cxctx->firstcomplex;
    cxctx->firstcomplex=complex;

    complex->re=complex->im=0;          /* Initialize the complex number */
    complex->refcnt=1;
    return complex;
  }
}
```

The *deletion function* frees the space used by a complex number and removes it from the list held by the module context (unless there still are references to this number, in which case the deletion function just decreases the reference counter).

7.2.3 Type transformation to and from string

To be able to use `initializations` blocks with the new type `complex` we define two functions for transforming the number into a string and initializing it from a string. The writing function is also used by the `write` and `writeln` procedures for printing this type. The reading function also gets applied when the type instance creation function is given a string.
The format of the string will obviously depend on the type. In this example the obvious format is "re+im*i*". The following function prints a complex number:

```
static int cx_tostr(XPRMcontext ctx, void *libctx, void *toprt, char *str,
                    int len, int typnum)
{
  s_complex *c;

  if(toprt==NULL)
  {
    strcpy(str, "0+0i");
    return 4;
  }
  else
  {
    c=toprt;
    return sprintf(str, "%g%+gi", c->re, c->im);
  }
}
```

The next function reads in a complex number from a string:

```
static int cx_fromstr(XPRMcontext ctx, void *libctx, void *toinit, const char *str,
                    int typnum)
{
  double re,im;
  s_complex *c;
  if(sscanf(str,"%lf%lf",&re,&im)!=2)
    return XPRM_RT_ERROR;
  else
  {
    c=toinit;
    c->re=re;
    c->im=im;
    return XPRM_RT_OK;
  }
}
```

7.2.4 Overloading of arithmetic operators

The only type conversion that is carried out automatically by Mosel is from integer to real (but not the other way round), and no conversions involving external types. It is therefore necessary to define all the operations between two numbers for two complex numbers and also for a complex and a real number. However, for commutative operations (addition, multiplication, comparison) it is only required to define one version combining the two types, the other sense is deduced by Mosel.

Taking the example of the *multiplication*, we have to define the multiplication of two complex numbers: $(a + bi) \cdot (c + di) = ac - bd + (ad + bd)i$

```
static int cx_mul(XPRMcontext ctx, void *libctx)
{
  s_complex *c1,*c2;
  double re,im;

  c1=XPRM_POP_REF(ctx);
  c2=XPRM_POP_REF(ctx);
  if(c1!=NULL)
  {
    if(c2!=NULL)
    {
      re=c1->re*c2->re-c1->im*c2->im;
      im=c1->re*c2->im+c1->im*c2->re;
      c1->re=re;
      c1->im=im;
    }
    else
      c1->re=c2->im=0;
  }
  cx_delete(ctx,libctx,c2,0);
  XPRM_PUSH_REF(ctx,c1);
  return XPRM_RT_OK;
}
```

and also the multiplication of a complex with a real: $(a + bi) \cdot r = ar + bri$

```
static int cx_mul_r(XPRMcontext ctx, void *libctx)
{
  s_complex *c1;
  double r;

  c1=XPRM_POP_REF(ctx);
  r=XPRM_POP_REAL(ctx);
  if(c1!=NULL)
  {
    c1->re*=r;
    c1->im*=r;
  }
  XPRM_PUSH_REF(ctx,c1);
  return XPRM_RT_OK;
}
```

It is not necessary to define the multiplication of a real with a complex since this operation is commutative and Mosel therefore deduces this case. The *addition* of two complex numbers and of a complex and a real number is implemented in a very similar way to multiplication. Once we have got the two types of addition, we simply need to implement the negation (–complex) in order for Mosel to be able to deduce *subtraction* (real – complex and complex – complex). For *division*, we need to implement all three cases since this operation is not commutative: complex/complex, complex/real and real/complex.

Furthermore we need to define the identity elements for addition and multiplication respectively:

```
static int cx_zero(XPRMcontext ctx, void *libctx)
{
  XPRM_PUSH_REF(ctx,cx_create(ctx,libctx,NULL,0));
  return XPRM_RT_OK;
}

static int cx_one(XPRMcontext ctx, void *libctx)
{
  s_complex *complex;

  complex=cx_create(ctx,libctx,NULL,0);
  complex->re=1;
  XPRM_PUSH_REF(ctx,complex);
  return XPRM_RT_OK;
}
```

Once addition and the 0-element have been defined, Mosel deduces the aggregate operator `SUM`. With multiplication and the 1-element, we obtain the aggregate operator `PROD` for our new type.

Other operators implemented by this module are constructors, the assignment and the comparison operators.

8 Conclusion

Mosel provides a flexible environment for modeling and solving optimization problems. The examples discussed in this paper show different possible uses of Mosel: practitioners in Operations Research typically are interested in implementing their models quickly and in a form that can easily be maintained. This is delivered by the fact that the formulation of mathematical models in the Mosel language is close to their algebraic form (see [Guéret et al., 2002] for a collection of examples). Mosel also provides ample support for data handling making it possible to read and write data from various sources with just a few lines of code. When models grow larger the possibility to write solution algorithms and heuristics directly in the Mosel language is often helpful. Advanced users and researchers appreciate the possibility to add themselves whatever new feature may be required by their application, such as access to specific external data sources, new solvers, or external solution algorithms.

The latter relies on an important characteristic of Mosel, its modular architecture: software- and/or application-specific functionality can easily been added in the form of modules that extend the Mosel language. This scheme goes so far as to allow modules to publish new data and variable types which may be required, for instance, to support solvers other than Linear and Mixed Integer Programming tools. These may be matrix-based solvers (such as for solving Quadratic Programming problems) or software that uses a different way of representing problems (like finite domain constraint solvers).

The Mosel environment with the various adjoint products of the FICO^{TM} Xpresssoftware suite covers the full cycle of application development, moving from the prototype stage (easily realized within Xpress-IVE) to the implementation of a complete optimization solution (developed, analyzed, and improved with tools such as the Mosel debugger, profiler, or the Xpress-Tuner) and its embedding into the company's information system (library interfaces), including connections to external data sources (databases, on-line data) or even the development of complete graphical application interfaces (XAD), without any need for changing the software platform at any point of the development process.

A comprehensive collection of examples is accessible from the Xpress examples database website:

`http://www.dashoptimization.com/home/cgi-bin/example.pl`

Bibliography

[Dash, 1999] Dash Associates. XPRESS-MP Reference Manual, 1999.

[Fourer et al., 1993] R. Fourer, D. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[Guéret et al., 2002] C. Guéret, S. Heipcke, C. Prins, M. Sevaux (2002). *Applications of Optimization with Xpress-MP*. Dash Optimization, Blisworth, UK.

[Maximal, 2001] Maximal Software. MPL for Windows Reference Manual, 2001.

[Van Hentenryck, 1998] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, 1998.