Mosel: An Extensible Environment for Modeling and Programming Solutions

Yves Colombani & Susanne Heipcke
Dash Optimization
Blisworth House
Blisworth, Northants NN7 3BX, U.K.
email: {yves.colombani,susanne.heipcke}@dashoptimization.com

Abstract
The aim of this paper is two-fold. Its first concern is to explain 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, the paper also shows how Mosel can be used to implement more complex solution algorithms. In the context of combining solvers that originate from different areas of research, the second interest of this paper is a different one: the open, modular architecture of the new environment for modeling and solving that is presented here has been designed to be easily extensible, not being restricted to a particular (type of) solver. The paper explains how the existing Mosel language may be extended to provide new functionality and access to other solvers.

1 Introduction
Mosel is a new 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 [2] for which the problem is described using a “modeling language” and algorithmic operations are written with a “scripting language” (similarly for OPL [5] 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.

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

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. We shall not enlarge on this topic since this would lead beyond the scope 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. Subsequently, we demonstrate how Mosel can be used to implement more advanced solution algorithms (column generation, recursion). 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. Last but not least, the solution of a non-linear problem gives an example of the joint use of different solvers (Linear Programming/Neural Networks) in Mosel. The paper concludes with some remarks about ongoing and future developments and hints at possible extensions of the Mosel language, especially with hindsight to finite domain solvers.

2 Overview

2.1 The Mosel Language: a Simple Example

```plaintext
model Chess
uses "mmpars"

declarations
xs,xl: mpvar
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 ! Integality 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
```

This 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 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.

**Solving:** With the procedure `maximize`, we call Xpress-Optimizer to maximize the linear expression $5 \times xs + 20 \times xl$. 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.

**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.

```mosel
model Chess2
uses "mmxprs"

declarations
Allvars: set of mpvar ! Set of variables
DescrV: array(Allvars) of string ! Descriptions of variables
xs,xl: mpvar ! x
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
```
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.

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 (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 types set (collection of elements of a given type) and array (collection of labeled objects of a given 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 static arrays (A1, A2):

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

Dynamic sets (R2, S2) 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
end-declarations
```

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

3.2 Initialization of Data/Data File Access

Dynamic arrays and sets are often used when data are read from a file 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
deadclarations
initializations from "initdata.dat"
A C S
end-initializations
The data file initdata.dat that is read by this program has the following contents:

A: [2 4 6 8]
C: ["red" 3 ("green") 4.5 ("blue") 6 ("yellow") 2.1]
S: ["white" "black"]

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:

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']

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:

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:

```mosel
declarations
    A : integer
    x: mpvar
end-declarations
if A >= 20 then
    x <= 7
elsif 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

```mosel
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-case
```

3.4 Loops

Loops regroup 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

```mosel
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 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), but it is also possible to define new procedures and functions
according to the needs in 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 behind the name of the subroutine.

```mosel
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 as well 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 as well as for his own functions and procedures.

### 4 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
- 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), 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.

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 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.

4.1 Available Modules

At the present, the following modules have been implemented:

**Solvers:** mmxprs, mmocl

**Data handling:** mmodbc, mmetc

**System:** mmsystem

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.

```mosel
uses "mmxprs"
declarations
 x: array(1..10) of mpvar
end-declarations
procedure UIS
 writeln("Solution: ", getsol(Objective))
 forall(i in 1..10) write("x(",i,")=",getsol(x(i))," |
 writeln
end-procedure

setcallback(XPRS_CB_UIS,"UIS")
```

The module mmocl provides access to the OptQuest Callable Library for solving non-linear problems (see the example in Section 5.2).

Modules may define additional interfaces to data files: mmetc defines the procedure diskdata that emulates the data in- and output of mp-model[1]; module mmodbc provides access to any data source for which an ODBC interface is available, using standard SQL commands. The following program extract reads a 2-dimensional array and its sizes from an MS-Excel spreadsheet:
model sizes
  uses "mmodbc"
end-declarations
SQLconnect('DSN=MSExcel;DBQ=ssxmpl.xls')
Nprod:=SQLreadinteger("select Nprod from SIZES")
Nrm :=SQLreadinteger("select Nrm from SIZES")
delclarations
  PneedsR: array(1..Nprod,1..Nrm) of real
delclarations
SQLexecute("select * from USAGE",[PneedsR])
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.

5 Examples

In this section we give two examples of solution algorithms that use specific functionality of the solver that is employed. All subroutines, constants and control parameters that are contributed to the Mosel language by the corresponding solver module are highlighted in bold face.

5.1 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.

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

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

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

    setparam("XPRS_PRESOLVE",0)
    maximize(XPRS_TOP,MaxProfit) ! Solve the LP problem
    savebasis(1) ! 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) < TOL) then
            setub(open(f,t), 0)
        elseif(1-osol(f,t) < TOL) then
            setlb(open(f,t), 1)
        end-if
    end-do

    maximize(MaxProfit) ! Solve the modified MIP
    solval:=getobjval

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

    loadbasis(1)
    setparam("XPRS_MIPABSCUTOFF", solval) ! Set "cutoff" value
    maximize(MaxProfit) ! Solve the original MIP
    returned:=getobjval
end-function
```
5.2 mmocl: Solving an NLP

OptQuest Callable Library (OCL) [4] is a solver for non-linear problems: linear problems with a non-linear objective function (NLP) or even problems with non-linear constraints that are expressed in the form of so-called “requirements” (lower/upper bounds on a function that maps values of decision variables to a real number).

The following example solves an NLP problem with 7 variables (three of which are constrained to take integer values only), bounds and linear inequality constraints on these variables, and the non-linear objective function

\[
\min -x_1^2 + (2 \cdot x_2 - x_3)^3 + (3 \cdot x_3 - 4 \cdot x_4) \cdot (x_6 - 5 \cdot x_5) - (x_6 - x_7)^2
\]

After the problem has been set up in OCL, the following loop is performed for a fixed number of iterations TotalIter:

– Obtain a solution from the solver
– Return the resulting value of the objective function to the solver

Several times during the execution of this loop and after its termination, the solution information held in Mosel is updated with the best solution found by the solver (procedure OCLGetBest).

```mosel
model "ocltest 2"
uses "mmocl"
parameters
   TotalIter=100
end-parameters
function objmin(x:array(range) of mpvar): real
   returned:= -getsol(x(1))^2 + (2*getsol(x(2)) - getsol(x(3)))^3 + (3*getsol(x(3)) - 4*getsol(x(4)))*getsol(x(6)) - 5*getsol(x(5)) - (getsol(x(6)) - getsol(x(7)))^2
end-function
declarations
   x:array(1..7) of mpvar
end-declarations
2*x(1) - x(2) - x(3) + x(4) - 2*x(5) + x(6) - x(7) = 40
2*x(1) + 2*x(2) - 3*x(3) + x(4) + 2*x(5) + 2*x(6) + x(7) >= 60
6*x(1) + 2*x(2) - 3*x(3) + 4*x(4) + x(5) + 2*x(6) <= 120
2*x(1) + x(2) + x(3) + 4*x(4) + x(5) + 2*x(6) >= 20
0 <= x(1); x(1) <= 10
-5 <= x(2); x(2) <= 20
2.5 <= x(3); x(3) <= 14.3
1 <= x(4); x(4) <= 15
0.5 <= x(5); x(5) <= 23.4
0 <= x(6); x(6) <= 30
0 <= x(7); x(7) <= 28
x(1) is_integer
x(6) is_integer
setstepsize(x(4),3) ! Integer variable with step size 3
setparam("OCLVerbose",true)
OCLSetup(OCL_MIN,[],[])
setparam("OCLSolutions",TotalIter) ! Set up the problem in OCL
```

OCLInitPop
forall(i in 1..TotalIter) do
  OCLGetSolution ! Update with new solution
  OCLPutSolution(objmin(x)) ! Return the evaluation of the solution
  if i mod 10 = 0 then
    OCLGetBest ! Update with best found solution
    writeln("Best solution after ",i," iterations is ",getobjval)
  end-if
end-do

writeln
OCLGetBest
forall(i in 1..7) writeln("x(",i,") = ",getsol(x(i)))
writeln("Objective is ",getobjval)
end-model

6 Mosel C Libraries

Models written with the Mosel language can be accessed from C through the Mosel C interface. 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 binary model (BIM) 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 */
7 Conclusion

Mosel provides a flexible environment for working with problems based on linear constraints. Its programming facilities make possible the implementation of complex solution algorithms directly in the Mosel language. An important characteristic of Mosel is 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 are required, for instance, to support finite domain constraint solvers.

Mosel has recently been commercialized under the name of Xpress-Mosel by Dash Optimization. For further information see http://www.dashoptimization.com.

References