dash optimization

mod2mos

mp-model to Mosel translator

User Guide

Release 1

How to Contact Dash

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

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 567 9445
Fax: (201) 567 9443
email: support-usa@dashoptimization.com

Elsewhere
Dash Optimization Ltd.
Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5RX
UK
Telephone: +44 1926 315862
Fax: +44 1926 315854
email: support@dashoptimization.com

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

USA, Canada and The Americas
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
Telephone: (201) 567 9445
Fax: (201) 567 9443
email: sales@dashoptimization.com

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

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com

Contents

Chapter 1
Introduction

The program **mod2mos** is intended to help in the process of migrating models written for mp-model to Mosel by translating source files from one syntax to the other.

The translation is operated in a way that as far as possible preserves the structure of the original file (location of comments, order of instructions).

Although in many cases the resulting file is ready for compilation with the Mosel compiler, some minor modifications may have to be carried out manually either before (on the original mp-model source file) or after the conversion.

This document describes the most frequent cases that may lead to difficulties and the corresponding modifications to remedy or prevent these. Many problems are detected either by the translator itself or by the Mosel Compiler. In addition, one section of this manual is dedicated to cases that cannot be detected by either one and only show up during the execution of the translated model. The last chapter contains suggestions as to how a generated Mosel code may be improved, in terms of readibility and efficiency.

Chapter 2
Using mod2mos

The mp-model to Mosel Translator is started with the command `mod2mos`. Depending on the set of parameters, the command may translate a model source file or a data file.

2.1 Processing model source files

mp-model source file: The command `mod2mos` expects as parameter the name of the file to be translated. If the file name is given without an extension and no file of this name exists, the extension `.mod` is assumed. If no argument is given, the source is taken from the standard input.

Generated output file: The program writes its result to a file the name of which is deduced from the input file name with extension `.mos` (if no filename was given the result is sent to the standard output).

A typical use of **mod2mos** is the following (assuming we wish to translate the file `myfile.mod` (mp-model syntax) to the file `myfile.mos`):

```
mod2mos myfile
```

The following options may be used with the `mod2mos` command:

-h: display a short help message

-V: display version information

-t *num*: set the size of tabulations to *num*. The default value for this parameter is 4 under Windows and 8 for other operating systems.

-e *ext*: use the extension *ext* when renaming an identifier. The default value for this parameter is '_'

-i*n*: during the conversion, **mod2mos** may also replace 'diskdata' statements by Mosel initialization blocks and optionally generate the corresponding data files based on the original mp-model files. The level of conversion is determined by *n* which is bit encoded as follows:

- **bit 1 (1):** translate input statements only
- **bit 2 (2):** translate output statements only
- **bit 3 (4):** by default data files are not modified and the command to be executed is displayed. If this flag is set, the command is executed
- **bit 4 (8):** disable printing of messages related to this option

-o *file*: use *file* as the output filename. If *file* is '-' the result is sent to the standard output.

Example:

```
mod2mos -t 6 -e \_\_ -i15 -o newmod.mos oldmod
```

2.2 Processing data files

mod2mos may also be used to convert a data file suitable for a diskdata statement into an initialization block file. This mode is activated by the option *-d*:

-d *fmt*: if this option is used, **mod2mos** expects a data file instead of a model file. Since mp-model data files are not structured, the parameter 'fmt' is required to indicate the structure to use and what the label(s) in the resulting file must be:

label1[,label2,... :] the file contains a (list of) scalar(s)

label/0: the file contains an array in dense format

label/i/n: the file contains *n* array(s) in sparse format and the *ith* first columns are used as indices

When the program is used in this mode, the file name produced for the output file is *original_name*_mos.dat. Note that the result of the conversion is appended to the end of the output file (normally the file is replaced).

Examples:

Assuming we need to translate the file `scs.dat` that contains the values for scalars `E1`,`E2`, `E3` and `E4`. Such a file may look like the following:

```
10,5,7.8,0
```

The conversion can be achieved by the following command:

```
mod2mos -d E1,E2,E3,E4 -o datainit.dat scs.dat
```

which produces the file:

```
E1: 10
E2: 5
E3: 7.8
E4: 0
```

The same `scs.dat` file may be employed to initialize the mp-model dense table `T`. In this case the command should be:

```
mod2mos -d T/0 -o datainit.dat scs.dat
```

which gives the file:

```
T: [
 10 5 7.8 0
]
```

An mp-model data file can also be used to initialize several arrays. In this case, the first columns of each line of the file are used as indices and the following data are the values to be assigned to the array cells. Assuming the file `mytabs.dat` is intended to work with the arrays `T1` and `T2` that are of 3 dimensions, the contents of such a file may be:

```
1,a,1,10,20
1,a,2,30,40
1,b,1,50,60
```

The conversion can be obtained with the following command:

```
mod2mos -d T12/3/2 -o datainit.dat mytabs.dat
```

and produces the following initialization file:

```
T12: [
 (1 a 1) [10 20]
 (1 a 2) [30 40]
 (1 b 1) [50 60]
]
```

Chapter 3
Cases requiring modifications

3.1 Problems detected during the translation

During the processing of an mp-model file, **mod2mos** may detect various cases where either it cannot produce valid code or the generated code is potentially incorrect. Usually a simple modification in the generated file fixes the problem.

Below we list the **mod2mos** error messages together with a description of the modifications required to counter the problem.

- ***Only ODBC connections are supported.***

  Currently, database access is achieved using the `mmodbc` module with Mosel (ADO is not supported).

- ***Conditional file inclusion is not supported.***

  mp-model treats its statements one at a time and the result of a statement can be used to select what to do next, using conditions for choosing the following statement(s), possibly ignoring a part of the source file. Mosel compiles a source file at once; conditions are evaluated at run time but file inclusion is performed at compile time.

  Example:
  ```
  IF X=0
   INPUT file1.mod
  ELSE
   INPUT file2.mod
  ENDIF
  ```

  This block of statements cannot be translated to Mosel: file inclusions have to be moved outside of conditions.

- ***Declarations in conditions are not supported.***

  The same restriction as for file inclusion applies to declarations:
  ```
  IF X=0
   TABLES T(10)
  ENDIF
  ```

  In mp-model, if the condition evaluates to false the symbol T is not declared. With Mosel this decision cannot be taken since the condition will be checked at execution time. A workaround is to move the condition to the size of the array.

  Example:
  ```
  declarations
   T:array(1..if(X=0,10,1)) of real
  end-declarations
  ```

  At execution time, if the condition is not satisfied, only a small array (1 element) is created. If the same symbol is used for incompatible objects, this workaround is not sufficient and it becomes necessary to rename the object.

Example:

```
IF X=0
 TABLES T(10)
ELSE
 TABLES T(10,20)
ENDIF
```

can be rewritten as follows:

```
declarations
 T1:array(1..if(X=0,10,1)) of real
 T2:array(1..if(X<>0,10,1),1..if(X<>0,20,1)) of real
end-declarations
```

As a general rule, such modeling techniques should have been avoided because they are a likely source of mistakes and make the model harder to read and maintain.

- ***Condition on declaration of 'x' ignored.***

  An expression of the following type has been encountered:

  ```
  VARIABLES x(|Z>0)
  ```

  Since conditions in declarations are not supported, this statement cannot be translated. To get the corresponding behaviour with Mosel, the condition may be translated into a constraint:

  ```
  declarations
   x:mpvar
  end-declarations
  if Z<=0 then x=0; end-if
  ```

- ***'diskdata' cannot be used to initialize scalars ('a').***

  The procedure `diskdata` of the module `mmetc` works only with arrays.
  The following mp-model code produces this warning:

  ```
  TABLES a
  DISKDATA a=myfile
  ```

  because it is translated to:

  ```
  declarations
   a: real
  end-declarations
  diskdata('myfile',a)
  ```

  which produces the following error in Mosel: *Incompatible types for parameters of 'diskdata'.*

  To avoid this problem, an auxiliary array may be introduced as an intermediate:

  ```
  declarations
   a: real
   a_in: array(1..1) of real
  end-declarations
  diskdata('myfile',a_in)
  a:=a_in(1)
  ```

  In the case of ODBC connections, the preferable solution is to use functions **SQLread...**; in our example:

  ```
  declarations
   a: real
  end-declarations
  a:=SQLreadreal('select * from T')
  ```

- ***'diskdata' does not support offset ('a').***

  Statements of the form:

```
DISKDATA a(10)=myfile
```

are not properly converted. Since the Mosel procedure `diskdata` accepts only arrays as parameter, either a separate operation is required to shift the data within the table or an intermediate table needs to be used. The line above may, for instance, be replaced by the following:

```
diskdata(ETC_IN,'myfile',a)
forall(i in 1..10) do
 a(i+9):=a(i)
 a(i):=0
end-do
```

- ***'initializations' does not support offset ('a').***

  This is the same problem as above but when initialisation blocks are generated from diskdata statements. In this case, the problem may be fixed by adding the offset in the data file directly:

```
a: [ (10) 1 2 3 4 ]
```

- ***'diskdata' option 'X' is ignored.***

  The implementation of `diskdata` in `mmetc` supports only the options *d*, *s*, *c*, *a*, *t*, *m*, *u*. Note further that `mmodbc` supports none of these.

- ***'select' must be replaced by 'insert' in SQL queries.***

  When saving data to databases using ODBC (post optimization), `select` SQL queries have to be replaced by `insert` or `update` queries because of the different way of working of `mmodbc` compared to ODBC-connect of mp-model.

  Note also that exporting arrays of variables or constraints is not supported by `mmodbc`: solution values have to be stored in data arrays for use in SQL queries.

  Example:

```
declarations
 sol_x:array(I) of real
end-declarations
forall(i in I) sol_x(i):=getsol(x(i))
SQLupdate('select * from solutions',sol_x)
```

- ***'nodynindex' has to be replaced by set finalizations.***

  The dynamic initialization of index sets is controled by the option `DYNINDEX` in mp-model. This feature does not exist in Mosel and `diskdata` behaves as if `DYNINDEX` was always active. Actually, as long as a set `I` in a Mosel model is dynamic, a request to an unknown element in `I` implies an extension of this set. To reproduce the effect of the statement `SET NODYNINDEX`, sets that have been initialized must be finalized.

  Example:

```
diskdata('datafile.dat',I)
finalize(I)                  ! 'I' is now constant
```

- ***Trying to convert macro 'm' into a procedure.***

  Since Mosel does not support macros, the translator tries to generate a procedure in place of the macro (and modifies the calls to this macro accordingly). This conversion does not work if the macro uses its parameters for building names or declares new entities (in Mosel they become local to the procedure).

  Example:

```
MACRO tut a,b
 LET a=10
 LET my b=0
ENDMACRO

tut aa,bb
PRINT aa
PRINT mybb
```

This code cannot be properly converted.

- ***Symbol 'a' (Mosel keyword) is renamed to 'a_'.***

  Reserved words in Mosel and mp-model are not the same. Some identifiers that are valid for mp-model are keywords in Mosel: in such a case, **mod2mos** renames the identifier by appending an underscore to the original name (the string to be appended to the identifier name can be modified by using the option '-e' of **mod2mos**).

  Example:

```
TABLES prod(10)
```

  results in:

```
declarations
 prod_:array(1..10) of real
end-declarations
```

- ***SLANG section cannot be handled. Aborting.***

  There is no one-to-one correspondence between SLANG and Mosel but most of the tasks programmed with SLANG in mp-model models can quite easily be rewritten directly with Mosel.

- ***Statement 's' ignored.***

  With *s = delete, columns, help, pause, show, toasc, save, restore, 123data*.

  These statements have no direct correspondence in Mosel and are therefore ignored during the conversion.

  Note that using procedures with locally declared data is a good alternative to deleting tables.

3.2 Problems detected by Mosel during compilation

Even when the translation has been performed without any problem, the generated code may not always be a correct Mosel program. Here are some typical cases where Mosel detects syntax errors that are due to specificities of mp-model.

- ***Incompatible types for range.***

  As opposed to mp-model, Mosel does not convert reals to integers implicitly. In many mp-model models, reals are used to define subscript ranges: in the context of Mosel, it is important to make sure that the values used to define an index range are indeed of type integer.

  For instance:

```
TABLES MM
...
ASSIGN C(i=1:MM)=1
```

  The translated code will not work because MM is not an integer. The correction can be done in the original model:

```
TABLES -i MM
```

  This error may also be produced when strings are used as range bounds.

  Example:

```
FOR(i="Mon":"Fri")...
```

Mosel supports only ranges of *integers* – in this example a *set* of strings is required. Moreover, arithmetic on index sets elements is not defined in Mosel since string operations are supported. For instance: `"Sat"-1` is interpreted as 'the element preceding `"Sat"` in the index set' in mp-model. In Mosel the string difference gets computed (that is, `"Sat"-"1"` which gives `"Sat"`).

As a consequence of the previous remarks, the following mp-model statement:

```
FOR(i="Mon":"Sat"-1)...
```

has to be rewritten for Mosel (assuming that `days` has been defined previously as the set of all days of the week):

```
forall(i in days-{"Sat","Sun"})...
```

- ***Expression cannot be assigned a value.***

  mod2mos translates `LET` statements into constants. However, sometimes `LET` statements are used in the same way as `ASSIGN` and the same symbol is assigned different values using consecutive `LET` statements. Starting from the second assignment, **mod2mos** translates this statement into Mosel assignments: since the symbol was originally defined as a constant, a compilation error is raised.

  Example:

```
LET a=10
LET a=20
```

  results in the Mosel code:

```
declarations
 a=10
end-declarations
a:=20
```

  In such a case, either the `LET` statements need to be replaced by `ASSIGN` statements in the mp-model source file or in the generated Mosel file the constant declaration must be replaced by a variable declaration.

  Example:

```
declarations
 a:integer
end-declarations
a:=10
a:=20
```

In certain cases, the Mosel Compiler displays the error message *'Syntax error'* without being able to give any more detailed information. This may be due to the following:

- **Syntax error related to parameters blocks**

  mod2mos converts `DEFINE` statements into model parameters. In mp-model defined objects are handled like macros, in Mosel they are constants. In some places this difference implies incorrect translations.

  Example:

```
DEFINE A=tut
TABLES t%A%t(10)
```

  gives:

```
parameters
 A=tut
end-parameters

declarations
 tAt: array (1..10) of real
end-declarations
```

- **Syntax error on 'diskdata' or 'SQLexecute' statements**

  If `diskdata` or ODBC statements that raised error messages during the translation have not been corrected, the Mosel Compiler is likely to stop with a syntax error on the corresponding statement. See the remarks concerning these topics in Section 3.1.

3.3 Problems that are not detected

It is important to test generated code with different data sets and compare results with the original model run through mp-model. The translated code may indeed be syntactically correct (*i.e.* it is compiled cleanly) without being semantically correct (*i.e.* it does not define exactly the same model as the source code). In such a case the generated model does not behave as expected. Here are some guesses for locating possible causes of the problems.

- **No variables are generated in the problem.**

  mp-model generates variables when they are used in constraints.

  Example:

```
INDICES I(10)
VARIABLES v(I)
...
DISKDATA I= datafile.dat
```

  This example is translated into:

```
declarations
 I: set of string
 v:array(I) of mpvar
end-declarations
...
diskdata('datafile',I)
```

  Since `I` is unknown when v is declared, the array is dynamic and no variable is created. To fix the problem, the declaration of v has to be moved *after* the initialization of `I`, possibly finalizing the set.

  Example:

```
declarations
 I: set of string
end-declarations
...
diskdata('datafile',I)

finalize(I)
declarations
 v:array(I) of mpvar
end-declarations
```

- **Data table not properly initialized through ODBC.**

  By default, `mmodbc` expects that the first columns of a data table represent the indices for the array to initialize. But in some cases, all columns have to be treated as data.

  Example:

```
T1: 1,2,3
    4,5,6
    7,8,9

TABLES T(3,3)
DISKDATA -c T='select * from T1'
```

Because of the default behavior of mmodbc, the initialization of T is not performed properly (we expect: T(1,1)=1, T(1,2)=2, T(1,3)=3, ... – and mmodbc reads: T(1,2)=3, T(4,5)=6 and produces an *out of range* error). The correct behavior is obtained by changing the value of the control parameter SQLndxcol before executing the SQL query.

Example:

```
setparam('SQLndxcol',false)
SQLexecute('select * from T1',T)
setparam('SQLndxcol',true)
```

Note that if the option 'd' is used with diskdata (in the mp-model source), the translator generates the correct statements. The fix can therefore be done on the original source:

Example:

```
DISKDATA -cd T='select * from T1'
```

- **String comparison**

  In mp-model all strings are 8 characters long, shorter strings are padded with spaces. As a consequence, the strings "a" and "a " are equal for mp-model but not for Mosel.

Chapter 4
Improving the generated code

Once an mp-model file has been correctly translated to a Mosel model and the resulting code has been validated, one may consider the conversion process to be completed. However, by taking advantage of the facilities provided by Mosel the generated code can be improved in various ways.

To ease the further use and development of the generated Mosel model, it may be helpful to take into account the following points:

- **Data file format**

  Data files formatted for being read with `diskdata` may be translated to the more flexible native format of Mosel read with `initializations` (with this format different data arrays may be contained in a single data file). Options *-i* and *-d* of `mod2mos` may be useful for this operation.

- **Structuring the model**

  Procedures and functions may be introduced to structure a model. Large model files could even be split into several files (using the `include` statement).

- **Using the Mosel language for problem solving and programming tasks**

  Optimization operations, solution heuristics, data preprocessing, evaluation and formated output of results can be included directly in the Mosel model file.

The following two simple modifications may have a dramatic impact on the readability and, perhaps more importantly, on the efficiency of the model.

- **Finalization of sets and dynamic arrays**

  In Mosel, an array is dynamic if it is indexed by a dynamic set. If an array is used to represent dense data, one should avoid defining it as a dynamic array that uses more memory and is slower than the corresponding static array. When translated directly, some mp-model models imply the use of dynamic arrays where these are not required.

  Example:

```
INDICES I(10)
TABLES A(I)
TABLES B(I)
DISKDATA A = datafile.dat
```

  gives in Mosel:

```
declarations
 I: set of string
 A: array(I) of real
 B: array(I) of real
end-declarations
diskdata('datafile.dat',A)
```

In this example A and I are initialized using a data file, so A needs to be dynamic (we do not know in advance its size – note that with mp-model the arrays are of the maximum size of I, here: 10). However, the second array could be declared static:

```
declarations
 I: set of string
 A: array(I) of real
end-declarations
diskdata('datafile.dat',A)
finalize(I)                 ! Fix the size of I

declarations
 B: array(I) of real        ! Now B is static
end-declarations
```

As a general rule, the following sequence of actions gives better results (in terms of memory consumption and efficiency):

1. declare data arrays and sets that are to be initialized from external sources,
2. perform initializations of data
3. finalize all related sets
4. declare any other arrays indexed by these sets (including decision variable arrays).

- **Projections are superfluous in Mosel**

  Projections are frequently used in mp-model files (section PROJECT) in order to improve the efficiency of models with very sparse data arrays. The major problem of projections is that their use makes the model source particularly hard to read and maintain.

  Mosel provides a `project` functionality that corresponds to the projections in mp-model and is used in the generated code. But in Mosel there is no need to use these projections since the compiler is able to identify sparse loops and optimizes them automatically. It is therefore good practice to remove the projections once the model has been translated to Mosel in order to improve its readability (and, in certain cases, its efficiency).

  Example:

```
TABLE A(1000,500) -e100
DISKDATA -s A= datafile.dat
PROJECT p=A
...
CONSTRAINTS C:SUM(i=1..entries(A)) A(p(i,1),p(i,2))*x(p(i,1),p(i,2)) = 0
```

  can be rewritten in Mosel:

```
declarations
 I=1..1000
 J=1..500
 A:dynamic array(I,J) of real
end-declarations
diskdata('datafile.dat',A)
...
C:=sum(i in I,j in J|exists(A(i,j))) A(i,j)*x(i,j) = 0
```

  For efficient use of the function `exists`, the following rules have to be observed:

  1. The arrays have to be indexed by named sets (here I and J).

     Example:

```
A: dynamic array(I,J) of real              ! can be optimized
B: dynamic array(1..1000,1..500) of real   ! cannot be optimized
```

  2. The same sets have to be used in the loops.

     Example:

```
forall(i in I,j in J | exists(A(i,j)))       ! fast
forall(i in I,j in 1..500 | exists(A(i,j)))  ! slow
```

3. The order of the sets has to be respected.

   Example:

```
forall(i in I,j in J | exists(A(i,j)))             ! fast
forall(j in J,i in I | exists(A(i,j)))             ! slow
```

4. The `exists` function calls have to be at the beginning of the condition.

   Example:

```
forall(i in I,j in I | exists(A(i,j)) and i+j<>10)  ! fast
forall(i in J,j in J | i+j<>10 and exists(A(i,j)))  ! slow
```

5. The optimization does not apply to `or` conditions.

   Example:

```
forall(i in I,j in J | exists(A(i,j)) and i+j<>10)  ! fast
forall(i in I,j in J | exists(A(i,j)) or i+j<>10)   ! slow
```

Index