Whitepaper

FICO® Xpress Optimization

Generalized file handling in Mosel

FICO® Xpress Optimization whitepaper

Last update 17 October, 2017

Generalized file handling in Mosel

Y. Colombani and S. Heipcke

Xpress Team, FICO, FICO House, Starley Way, Birmingham B37 7GN, UK
http://www.fico.com/xpress

17 October, 2017

Abstract

This paper describes the I/O drivers of the Mosel distribution with examples of their use. With only minimal changes to his models the user may switch between data sources of different formats. The interaction and exchange of data between a model and the application executing it can be made more immediate and as a consequence, more efficient. It is also possible to avoid the creation of physical intermediate files by performing all operations in memory. The latter may be useful, for instance, in distributed applications.
With the help of the Mosel Native Interface (NI) the user may also implement his own drivers. We introduce some applications of this feature: a driver for working with compressed files and an example generating C code with a Mosel model.

Contents

1 Introduction

Mosel works with a generalization of the notion 'file'. In statements of the Mosel language (such as `initializations from/to`, `fopen`, and `fclose`) and in the Mosel library functions (*e.g.* `XPRMcompmod`, `XPRMloadmod`, or `XPRMsetdefstream`) the term 'file' is not limited to 'physical file handled by the operating system'. 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 type of the file is indicated by an extended file name: the actual file name is preceded by the name of the driver that is to be used to access it. We now write, for example, `mem:filename` to indicate that we work with a file held in memory, or `mmodbc.odbc:mydata.mdb` to access an MS Access database via the ODBC driver provided by the module *mmodbc*. The default driver (no driver prefix) is the standard Mosel file handling. The general form of an extended file name is

driver_name:file_name or
module_name.driver_name:file_name

In the first case, the driver is defined by the Mosel core, the second form means that the driver is provided by the module *module_name*. The structure of the *file_name* part of the extended file name is specific to the driver, it may also consist of yet another extended file name (*e.g.* `zlib.gzip:tmp:myfile.txt`).

This paper describes the I/O drivers of the Mosel distribution with examples of their use. Some drivers provide interfaces to specific data sources (such as ODBC). Others 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.
With the help of the Mosel Native Interface (NI) the user may also implement his own drivers. We introduce some applications of this feature: a driver for working with compressed files and an example generating C code with a Mosel model.

2 Overview of I/O drivers

I/O drivers defined by the Mosel core:

Driver	Description	Notes
bin	Write (and read) data files in a platform independent binary format	See example in Section 7
cb	Use a (callback) function as a file	See example in Section 6.3
mem	Use memory instead of physical files for reading or writing data	See example in Section 6.1
null	Disable a stream	See example in Section 5.3
raw	Implementation of the `initializations` block in binary mode (data exchange with a C host application)	See example in Section 6.2
sysfd	Working with operating system file descriptors	See example in Section 6.4
tee	Output into up to 6 files simultaneously	See example in Section 5.2
tmp	Extension to the default driver that locates the specified file in Mosel's temporary directory	See example in Section 5.1

Drivers defined by modules of the Mosel distribution (documented in the *Mosel Language Reference* under the corresponding module or in the individual manuals as indicated):

Driver	Description	Notes
deploy.csrc	Generate a C file that embeds the compiled model	See note in Section 8.2
deploy.exe	Generate a standalone executable to run a model	See note in Section 8.2
matlab.mws	Data exchange in memory with the MATLAB workspace	See example in the 'Xpress MATLAB interface' manual
mmetc.diskdata	Access data in text files in diskdata format	See example in Section 4.3
mmhttp.url	Access a file on an HTTP server via its URL	See example in Section *mmhttp* of the Mosel Language Reference Manual
mminsight.scenariodata	Inter-scenario data access within Insight apps	See Chapter *Inter-Scenario Data Access* of the 'Xpress Insight Mosel Interface Reference Manual'
mmjobs.shmem	Data exchange between several models using shared memory	See example in Section 6.1.3
mmjobs.mempipe	Use memory pipes for data exchange between models	See the whitepaper 'Multiple models and parallel solving with Mosel' for examples of *mmjobs* drivers
mmjobs.rcmd	Starts the specified command in a new process and connects its standard input and output streams to the calling Mosel instance	
mmjobs.rmt	Accessing physical files on a remote Mosel instance	
mmjobs.xsrv	Connect to a (remote) host running the Mosel Remote Launcher *xprmsrv*	
mmjobs.xssh	Secure version of the *xsrv* driver	
mmoci.oci	Access an Oracle database for reading and writing in `initializations` blocks	See the whitepaper 'Using ODBC and other database interfaces with Mosel' for examples
mmodbc.odbc	Access data in external data sources via an ODBC connection in `initializations` blocks	Requires presence of datasource-specific ODBC driver; see example in Section 4.1

mmsheet.csv	Access spreadsheets in CSV format in `initializations` blocks	
mmsheet.excel	Access data in MS Excel spreadsheets directly in `initializations` blocks	Windows platforms only; see example in Section 4.2
mmsheet.xls	Access spreadsheets in Excel's XLS format in `initializations` blocks	Under Windows, Linux, Mac
mmsheet.xlsx	Access spreadsheets in Excel's XLSX and XLSM formats in `initializations` blocks	Like *mmsheet.xls*
mmssl.base64	Ascii to binary converter	See Section *mmssl* of the Mosel Language Reference Manual
mmssl.crypt	Encryption/decryption using a symetric algorithm	
mmssl.hex	Binary to ascii converter	
mmsystem.pipe	Open a pipe and start an external program which as input or output stream	
mmsystem.text	Use a string or text object as a file	See example in Section 4.3.3
r.rws	Data exchange in memory with the R workspace	See example in Section *R* of the Mosel Language Reference Manual
zlib.deflate	Handles files compressed using the *zlib* compression format	
zlib.gzip	Handles files compressed using the *gzip* compression format	Implementation discussed in Section 8.1
zlib.zip	Access a file stored in a *zip* archive (read only)	

I/O drivers defined by the Mosel library interfaces:

Driver	Description	Notes
mmdotnet.dotnet	Use a C# stream or object in place of a file	See example in Section 6.3.2
mmdotnet.dotnetraw	Data exchange between a Mosel model and a C# application	C# version of *raw*; see example in Section 6.2.2
mmjava.java	Use a Java stream or a `ByteBuffer` in place of a file	See examples in Sections 6.1.1 and 6.3.1
mmjava.jraw	Data exchange via shared memory between a Mosel model and a Java application	Java version of *raw*; see example in Section 6.2.1

2.1 Displaying the available I/O drivers

The Mosel core drivers can be displayed from the command line with the following command (the listing will also include any drivers that are provided by currently loaded modules):

```
mosel exam -i
```

The drivers provided by modules are displayed by the `exam` command for the corresponding module (in this example: *mmodbc*)

```
mosel exam -i mmodbc
```

Library drivers (in particular the Java module *mmjava* that is embedded in the Mosel core and also the *mmdotnet* module on Windows platforms) can be displayed with the help of the corresponding program `mmdispdso.[c|cs|java]` in the subdirectory `examples/mosel/Library` of the Xpress distribution. The command for running the Java version might look as follows under Unix (please refer to the provided `makefile` for compilation of the example):

```
java -classpath $XPRESSDIR/lib/xprm.jar:. mmdispdso mmjava
```

or from a Windows command prompt:

```
java -classpath "%XPRESSDIR%\lib\xprm.jar;." mmdispdso mmjava
```

3 Example problem

The following small *knapsack problem* will be used as an example in the next sections.

A burglar sees 8 items, of different worths and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum WTMAX he can carry.

We introduce binary variables $take_i$ for all i in the set of all items (*ITEMS*) to represent the decision whether item i is taken or not. $take_i$ has the value 1 if item i is taken and 0 otherwise. Furthermore, let $VALUE_i$ be the value of item i and $WEIGHT_i$ its weight. A mathematical formulation of the problem is then given by:

$$\text{maximize} \sum_{i \in ITEMS} VALUE_i \cdot take_i$$

$$\sum_{i \in ITEMS} WEIGHT_i \cdot take_i \leq WTMAX \quad \text{(weight restriction)}$$

$$\forall i \in ITEMS : take_i \in \{0, 1\}$$

The objective function is to maximize the total value, that is, the sum of the values of all items taken. The only constraint in this problem is the weight restriction.

The following Mosel model (`burglar2.mos`) implements this problem. In this implementation data are read from the text file `burglar.dat`. Note that the decision variables are declared after the initialization of the data, that is, at a point in the model where the contents of the set `ITEMS` is known. If an array is declared before its index sets are known it is created as a dynamic array. Dynamic arrays of basic types (such as `VALUE` and `WEIGHT`) grow by adding data entries, but for dynamic arrays of decision variables the entries need to be created explicitly once the index sets are known. By moving the declaration of the variables after the initialization of the data we can avoid this.

```
model Burglar2
 uses "mmxprs"

 declarations
  WTMAX = 102                    ! Maximum weight allowed
  ITEMS: set of string           ! Index set for items
  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items
```

```
 end-declarations

 initializations from "burglar.dat"
  [VALUE,WEIGHT] as "BurgData"
 end-initializations

 declarations
  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the MIP-problem

! Solution output
 forall(i in ITEMS) SOLTAKE(i):= getsol(take(i))

 writeln("Solution:\n Objective: ", getobjval)
 writeln(SOLTAKE)

 initializations to "burglarout.txt"
  SOLTAKE
 end-initializations

end-model
```

The data file `burglar.dat` read by this model has the following contents.

```
! Data file for 'burglar2.mos'
          ! Item      Value  Weight
BurgData: [(camera)    [ 15   2]
           (necklace)  [100  20]
           (vase)      [ 90  20]
           (picture)   [ 60  30]
           (tv)        [ 40  40]
           (video)     [ 15  30]
           (chest)     [ 10  60]
           (brick)     [  1  10]]
```

The solution is output to the file `burglarout.txt` via `initializations to`. This prints the array `SOLTAKE` to a text file in `initializations` format:

```
'SOLTAKE': [('camera') 1 ('necklace') 1 ('vase') 1 ('picture') 1 ('tv') 0
 ('video') 1 ('chest') 0 ('brick') 0]
```

4 Data source access from Mosel models

Access to data files with formats other than the `initializations` text file format can be achieved with the help of modules or packages that define additional functions specific to a format or data source type. Alternatively, other formats can be used directly in the `initializations` block, and hence minimizing the changes required in a model to switch between different data formats.

The drivers *odbc*, *excel* and *diskdata* described in this section are used in the `initializations` block of the Mosel language to provide access to new data file formats and data sources. Some of

the drivers described in Section 6 (*e.g. raw* and *mem*) also provide such a functionality. However, they are only of use in models embedded into an application and run through the Mosel libraries, not in stand-alone Mosel models as is the case with the *odbc*, *excel* and *diskdata* drivers.

For a more in depth discussion of the database and spreadsheet drivers and examples of their use, the reader is referred to the Xpress whitepaper *Using ODBC and other database interfaces with Mosel*. Besides the ODBC and spreadsheet access functionality described in the following sections, this whitepaper also explains how to use the direct Oracle interface defined by the module *mmoci* (including the I/O driver *mmoci.oci* that behaves similar to *mmodbc.odbc* and SQL statements corresponding to the functionality described in Section 4.1.1) in the place of an ODBC connection.

4.1 Database access via ODBC

The *odbc* I/O driver defined by the module *mmodbc* automatically generates the SQL queries that are required to read in data from an external data source (database) or insert data into this data source. The following modifications need to be made to model *Burglar2* to switch to file access via ODBC:

```
model "Burglar2 (ODBC)"
 uses "mmxprs"

 parameters
  CNCTIO = ''
 end-parameters
 ...
 initializations from CNCTIO
  [VALUE,WEIGHT] as "BurgData"
 end-initializations
 ...
! Insert solutions into database: results from previous runs must be
! removed previously; otherwise the new results will either be appended
! to the existing ones or, if 'ITEM' has been defined as key field
! in a database, the insertion may fail.
 initializations to CNCTIO
  SOLTAKE as "SolTake"
 end-initializations
end-model
```

The *connection string* `CNCTIO` indicates the data source, for example:

- `CNCTIO = "mmodbc.odbc:burglar.mdb"` for the MS Access database `burglar.mdb`
- `CNCTIO = "mmodbc.odbc:DSN=mysql;DB=burglar"` for the mysql database `burglar`

To run this example, the ODBC driver for the corresponding data source must be present. The connection string may vary depending on the installation of the ODBC driver. For further detail on setting up an ODBC connection and working with ODBC in Mosel see the Xpress Whitepaper *'Using ODBC and other database interfaces with Mosel'*.

In the case of database connections, the data are read from a table called `BurgData` that contains (at least) a field for the indices (*e.g.*, labeled `Item`), and the fields `Val` and `Wght`. The results are written into the table `SolTake`, containing a field for the indices (*e.g.*, `Item`) and a field to receive the solution values (*e.g.*, called `IfTake`). In a spreadsheet, `BurgData` and `SolTake` are the names of named ranges. The columns of the named ranges must bear headers (to be included in the selected range area), these headers are not used by the (generated) SQL commands and may therefore differ from the names used in the model. The range `BurgData` must have three columns, for instance bearing the headers `Item`, `Val`, and `Wght`, and the range `SolTake` must have a column to receive the indices (*e.g.*, labeled `Item`) and a second one to receive the solution values (*e.g.*,

IfTake). If a database table has additional fields, or its fields do not appear in the order expected by Mosel, we can append the database field names to the table name, for example:

```
initializations from CNCTIO
 [VALUE,WEIGHT] as "BurgData(Item,Val,Wght)"
end-initializations
```

The ODBC driver may take several options, see the section '*mmodbc*' in the *'Mosel Language Reference Manual'* for further detail.

4.1.1 Alternative implementation using SQL statements

The model *Burglar2* may be implemented as follows using standard *mmodbc* functionality.

```
model "Burglar2 (SQL)"
 uses "mmxprs", "mmodbc"

 parameters
  CNCT = ''
 end-parameters
 ...
! Reading data from file
 SQLconnect(CNCT)
 SQLexecute("select * from BurgData", [VALUE,WEIGHT])
 SQLdisconnect
 ...
! Solution output
 SQLconnect(CNCT)
 SQLexecute("delete from SolTake")    ! Cleaning up previous results: works
                                      ! only for databases, cannot be used
                                      ! with spreadsheets (instead, delete
                                      ! previous solutions directly in the
                                      ! spreadsheet file)
 SQLexecute("insert into SolTake values (?,?)" , SOLTAKE)
 SQLdisconnect
end-model
```

As before, the connection string CNCT indicates the data source:

- CNCT = "burglar.mdb" (the full form "DSN=MS Access Database;DBQ=burglar.mdb" is determined automatically) for the MS Access database burglar.mdb
- CNCT = "DSN=mysql;DB=burglar" for the mysql database burglar

All of these functions may be used in conjunction with the ODBC I/O driver. For instance, when working with a database it may be helpful to make the lines

```
SQLconnect(CNCT)
SQLexecute("delete from SolTake")
```

preceed the insertion of the solution into the database to clean up previous results.

4.2 Spreadsheet drivers

The model *mmsheet* provides a set of spreadsheet I/O drivers that implement all basic data access tasks similarly to the functionality provided by ODBC (but none of the advanced SQL statements).

- *mmsheet.excel* is a software-specific I/O driver that accesses directly Excel spreadsheets. If the spreadsheet file is kept opened while running the Mosel model the output is written to the spreadsheet without saving it. This driver cannot be used remotely.
- *mmsheet.xls* and *mmsheet.xlsx* are I/O drivers for accessing spreadsheets in the Excel formats XLS and XLSX/XLSM respectively. They do not depend on Excel and can be used for reading and writing to spreadsheets on non-Windows platforms (Linux, Mac). They take the same options as *mmsheet.excel* and their use is analogous to this driver, with the exception that they save data immediately into the spreadsheet file, that is, an output file should not be open in another application while writing to it with these drivers.
- *mmsheet.csv* provides access to spreadsheet files in CSV format (text files). It is available for all platforms running Mosel and the file name is not restricted to physical files, it may be an extended file name. As with *xls/xlsx* output data is saved directly into the file.

With all *mmsheet* spreadsheet drivers output always starts at the same place (that is, previous output gets overwritten).

```
model "Burglar2 (Excel)"
 uses "mmxprs"

 parameters
  CNCTEXC = 'mmsheet.excel:burglar.xls'
 end-parameters
 ...

! Spreadsheet range includes header line -> option 'skiph' to skip header
 initializations from CNCTEXC
  [VALUE,WEIGHT] as "skiph;BurgData"
 end-initializations
 ...

! Insert solutions into spreadsheet: results from previous runs
! are overwritten by new output
! Only first line of output range is specified -> option 'grow'
 initializations to CNCTEXC
  SOLTAKE as "skiph;grow;SolTake"
 end-initializations
end-model
```

The input and output ranges in the spreadsheet used with the *excel* driver contain just the data, no column headers. To work with the ODBC format of ranges the option `skiph` needs to be used as shown in the example. Another option employed when outputting data is `grow`, this indicates that the output range is specified by a single row and may grow on demand.

Instead of naming the ranges in the spreadsheet it is equally possible to work directly with the cell references for the input and output ranges (including the worksheet name, which is '`burglar`' in our case):

```
 initializations from CNCTEXC
  [VALUE,WEIGHT] as "[burglar$B4:D11]"
 end-initializations
 ...

! Insert solutions into spreadsheet: results from previous runs
! are overwritten by new output
! Only first line of output range is specified -> option 'grow'
 initializations to CNCTEXC
  SOLTAKE as "grow;[burglar$F4:G4]"
 end-initializations
```

Or alternatively, using the *row-column notation*:

```
initializations from CNCTEXC
 [VALUE,WEIGHT] as "[burglar$R4C2:R11C4]"
end-initializations
...

initializations to CNCTEXC
 SOLTAKE as "grow;[burglar$R4C6:R4C7]"
end-initializations
```

Other formatting options for the specification of spreadsheet ranges include the *selection of columns* via their relative position within the cell range:

```
initializations from CNCTEXC
 [VALUE,WEIGHT] as "[burglar$A4:F11](#2,#3,#4)"
end-initializations
```

and the possibility to *omit row numbers*, thereby indicating that we want to read the contents of the entire column, typically used in combination with option `skiph` to skip the first row (assuming it contains the header text):

```
initializations from CNCTEXC
 [VALUE,WEIGHT] as "skiph;[burglar$F:G]"
end-initializations
```

Switching to the more generic driver *mmsheet.xls* in the examples above simply means changing the driver name in the filename string `CNCTEXC`. For CSV format, besides changing the driver name, we need to save the spreadsheet with CSV format. CSV files do not contain worksheets or range names, so ranges always need to be defined via cell references when using *mmsheet.csv*.

For further detail on the spreadsheet drivers the reader is refered to the section '*mmsheet*' of the *'Mosel Language Reference Manual'* and the examples discussed in the Xpress Whitepaper *'Using ODBC and other database interfaces with Mosel'*.

4.3 Alternative text data file formats

In the previous section we have introduced the I/O driver *mmsheet.csv* that reads and writes text files in CSV format. With only minimal changes to the previous model we now switch to a different text data format, namely the diskdata file format that is accessed via the *diskdata* I/O driver.

```
model Burglar2dd
 ...
 initializations from "mmetc.diskdata:sparse;noq"
  [VALUE,WEIGHT] as "burglardd.dat"
 end-initializations
 ...
 initializations to "mmetc.diskdata:append,sparse"
  SOLTAKE as "burglarout.txt"
 end-initializations
end-model
```

This will result in exactly the same behaviour as when using the procedure `diskdata` of the module *mmetc* instead of the `initializations` blocks.

```
model Burglar2ddb
 uses "mmetc", "mmxprs"
```

```
 ...
! Reading data from file
 diskdata(ETC_SPARSE+ETC_NOQ, "burglardd.dat", [VALUE,WEIGHT])
 ...
! Solution output
 diskdata(ETC_OUT+ETC_APPEND+ETC_SPARSE, "burglarout.txt", SOLTAKE)
end-model
```

The data file `burglardd.dat` used in both cases has the following contents.

```
! Data file for 'burglar2dd.mos'
camera,    15,  2
necklace, 100, 20
vase,      90, 20
picture,   60, 30
tv,        40, 40
video,     15, 30
chest,     10, 60
brick,      1, 10
```

The result file `burglarout.txt` is now formatted as shown here.

```
"camera",1
"necklace",1
"vase",1
"picture",1
"tv",0
"video",1
"chest",0
"brick",0
```

The *diskdata* I/O driver and the procedure `diskdata` are documented in the section '*mmetc*' in the 'Mosel Language Reference Manual'.

4.3.1 XML and JSON format files

The data for our example problem could also be provided in the form of the following XML file `burglar.xml`.

```
<!-- Data file for 'burglar2x.mos' -->
<BurgData>
  <Item name="camera">   <Value> 15</Value> <Weight> 2</Weight> </Item>
  <Item name="necklace"> <Value>100</Value> <Weight>20</Weight> </Item>
  <Item name="vase">     <Value> 90</Value> <Weight>20</Weight> </Item>
  <Item name="picture">  <Value> 60</Value> <Weight>30</Weight> </Item>
  <Item name="tv">       <Value> 40</Value> <Weight>40</Weight> </Item>
  <Item name="video">    <Value> 15</Value> <Weight>30</Weight> </Item>
  <Item name="chest">    <Value> 10</Value> <Weight>60</Weight> </Item>
  <Item name="brick">    <Value>  1</Value> <Weight>10</Weight> </Item>
</BurgData>
```

Given that XML files do not use any fixed format that would make it possible to define a suitable I/O driver, we cannot read or write XML files through `initializations` blocks. XML files need to be parsed or queried for their contents using the routines of the *mmxml* module. The data file `burglar.xml` is read into our Mosel data structures using the following code:

```
 declarations
  ITEMS: set of string           ! Index set for items
  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items
```

```
  BurgData, ResData: xmldoc      ! XML document
  Root, Node: integer            ! XML nodes
  NodeList: list of integer
 end-declarations

! Reading data from an XML file
 load(BurgData, "burglar.xml")

! Retrieve all 'Item' nodes
 getnodes(BurgData, "BurgData/Item", NodeList)

! Retrieve 'Value' and 'Weight' information
 forall(i in NodeList) do
   VALUE(getstrattr(BurgData,i,"name")):=
     getrealvalue(BurgData, getnode(BurgData, i, "Value") )
   WEIGHT(getstrattr(BurgData,i,"name")):=
     getrealvalue(BurgData, getnode(BurgData, i, "Weight") )
 end-do
```

For generating solution output in XML format we need to build up an XML document following standard XML design rules (all elements are forming a tree under a single root element).

```
! Create solution representation in XML format
 Root:=addnode(ResData, 0, XML_ELT, "SolTake")    ! Create root node "SolTake"
 setattr(ResData, Root, "objective", getobjval)   ! ... with attr. "objective"

 forall(i in ITEMS) do
   Node:=addnode(ResData, Root, XML_ELT, "Item") ! Add a node to "SolTake"
   setattr(ResData, Node, "name", i)             ! ... with attribute "name"
   setvalue(ResData, Node, SOLTAKE(i))           ! ... and solution value
 end-do

 save(ResData, "burglarout.xml")    ! Save solution to XML format file
```

This Mosel code will result in the following output (the appearance can be influenced via different controls and settings of *mmxml*, see the corresponding chapter of the *'Mosel Language Reference Manual'*):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SolTake objective="280">
  <Item name="camera">1</Item>
  <Item name="necklace">1</Item>
  <Item name="vase">1</Item>
  <Item name="picture">1</Item>
  <Item name="tv">0</Item>
  <Item name="video">1</Item>
  <Item name="chest">0</Item>
  <Item name="brick">0</Item>
</SolTake>
```

The module *mmxml* also provides routines for reading and writing text files in JSON format, transforming them internally to an XML-based representation where JSON 'values' are encoded as `jsv` elements. The value of the attribute 'jst' indicates the object type ('num'—number, 'boo'—boolean, 'str'—string, 'nul'—null, 'obj'—object, or 'arr'—array). Elements representing object members have the name of the member (instead of `jsv`).
Data from the following JSON file

```
{
  "Item": [
    {"Name":"camera",   "Value": 15, "Weight": 2},
    {"Name":"necklace", "Value":100, "Weight":20},
    {"Name":"vase",     "Value": 90, "Weight":20},
```

```
    {"Name":"picture",  "Value": 60, "Weight":30},
    {"Name":"tv",       "Value": 40, "Weight":40},
    {"Name":"video",    "Value": 15, "Weight":30},
    {"Name":"chest",    "Value": 10, "Weight":60},
    {"Name":"brick",    "Value":  1, "Weight":10} ]
}
```

can be read with this Mosel code (all declarations remain the same as in the previous model version reading XML format data):

```
! Reading data from a JSON file
 jsonload(BurgData, "burglar.json")

! Retrieve all 'Item' nodes
 getnodes(BurgData, "jsv/Item/jsv", NodeList)

! Retrieve 'Value' and 'Weight' information
 forall(i in NodeList) do
   VALUE(getstrvalue(BurgData, getnode(BurgData,i,"Name"))):=
     getrealvalue(BurgData, getnode(BurgData, i, "Value") )
   WEIGHT(getstrvalue(BurgData, getnode(BurgData,i,"Name"))):=
     getrealvalue(BurgData, getnode(BurgData, i, "Weight") )
 end-do
```

After loading the contents of a JSON file, it may be helpful to display the resulting XML representation on screen since we parse this representation to retrieve the input data:

```
 save(BurgData, "")        ! Display the XML representation on screen
```

The results output in JSON format is constructed using XML routines:

```
! Create solution representation in JSON format
 Root:=addnode(ResData, 0, XML_ELT, "jsv")            ! Create root node "jsv"
 Node:=addnode(ResData, Root, "Objective", getobjval) ! Add a node to "jsv"
 Node:=addnode(ResData, Root, XML_ELT, "Items")       ! Add a node to "jsv"
 forall(i in ITEMS)
   n:=addnode(ResData, Node, i, SOLTAKE(i))           ! Add a node to "Items"

 jsonsave(ResData, "burglarout.json")    ! Save solution to JSON format file
```

This results in the following output in file `burglarout.json`:

```
{
  "Objective":280,
  "Items":{
    "camera":1,
    "necklace":1,
    "vase":1,
    "picture":1,
    "tv":0,
    "video":1,
    "chest":0,
    "brick":0
  }
}
```

4.3.2 Free format files

If none of the provided text file access methods fits a particular data file format, you can always use `read`/`readln` in combination with `fopen` and `fclose`. For example, a data file of the form

```
! Data file for 'burglar2ff.mos'
camera [v=15, w=2]
necklace [v=100, w=20]
...
```

can be read by the following Mosel code (model file `burglar2ff.mos`)

```
fopen("burglarff.dat", F_INPUT) ! Open file for reading
while (not iseof) do            ! Read up to end-of-file
  fskipline("!")                ! Skip lines starting with '!'
  readln(j, " [v=", VALUE(j), ", w=", WEIGHT(j), "]")
  if getparam("nbread") < 6 then
    writeln("Error reading data for index '", j, "'")
  end-if
end-do
fclose(F_INPUT)                 ! Close the input file
```

And similarly, for redirecting output to a file instead of displaying it on the default output (screen) we simply surround it by `fopen` and `fclose`, this time selecting the output stream in `fopen`:

```
fopen("burglarsol.txt", F_OUTPUT)
writeln("Solution:\n Objective: ", getobjval)
forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
fclose(F_OUTPUT)
```

4.3.3 Multiline text or string

It is also possible to include data in the form of text blocks directly in a model. A multiline text (or string) must be surrounded by backquotes and an optional marker text. For example, the text data file read by the initial version of our model can be included and read thus (model version `burglar2t.mos`):

```
public declarations
 default_data: text            ! Data input values
end-declarations

default_data:=
`--------------- Input data ----------------
           ! Item      Value  Weight
BurgData: [(camera)    [ 15   2]
           (necklace)  [100  20]
           (vase)      [ 90  20]
           (picture)   [ 60  30]
           (tv)        [ 40  40]
           (video)     [ 15  30]
           (chest)     [ 10  60]
           (brick)     [  1  10]]
--------------- Input data ----------------`

initializations from "text:default_data"
 [VALUE,WEIGHT] as "BurgData"
end-initializations
```

And similarly, for redirecting output in Mosel's text format to a text block instead of writing it to a physical file we can use the following Mosel code:

```
public declarations
 burglarout: text              ! Solution output
end-declarations
```

```
 initializations to "text:burglarout"
  SOLTAKE
 end-initializations

! Display contents of output file
 writeln("Solution file:\n", burglarout)
```

5 Utilities and shorthands

This section lists several drivers that can be used from the Mosel libraries and also within Mosel models. These drivers are described in the 'Mosel Language Reference Manual'.

5.1 `tmp`: accessing the temporary directory of Mosel

The driver *tmp* is an extension to the default driver that locates the specified file in the temporary directory used by Mosel. For example, we might wish to compile a submodel file to a BIM file located in the temporary directory:

```
if compile("", "burglar.mos", "tmp:burglar.bim")<>0 then exit(1); end-if
load(modBurg, "tmp:burglar.bim")     ! Load the BIM file
```

The same can be achieved with the following library commands (Mosel C libraries):

```
XPRMcompmod(NULL, "burglar.mos", "tmp:burglar.bim", "Burglar example");
mod = XPRMloadmod("tmp:burglar.bim",NULL);     /* Load the BIM file */
```

The *tmp* driver can also be used with the library interfaces to other programming languages, such as the Mosel Java library:

```
 mosel.compile("", "burglar2.mos", "tmp:burglar2.bim", "");
 mod = mosel.loadModel("tmp:burglar2.bim");
```

Mosel's temporary directory is located in the system temporary directory, its actual definition depends on the system that is used for running Mosel and can be retrieved via the Mosel function `getparam`.

```
writeln("Temporary directory: ", getparam("tmpdir") )
```

5.2 `tee`: multiple output destinations

The *tee* driver duplicates output into up to 6 files simultaneously. It is particularly useful for creating output log files while still being able to watch progress output on screen. If we wish to maintain screen display and at the same time log the output in the file `burglarsol.txt` for the example `burglar2ff.mos` from Section 4.3 we may use the following Mosel code:

```
fopen("tee:burglarsol.txt&", F_OUTPUT+F_APPEND)
writeln("Solution:\n Objective: ", getobjval)
forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
fclose(F_OUTPUT)
```

The *tee* driver also works with the library functions, such as to duplicate all the output generated by a model (output to a file and on screen, Mosel C libraries):

```
XPRMsetdefstream(mod, XPRM_F_WRITE, "tee:burglarlog.txt&");
```

Analogously with other library interfaces, such as the Mosel Java library:

```
mod.setDefaultStream(XPRM.F_OUTPUT, "tee:burglarlog.txt&");
```

5.3 `null`: disactivating a stream

The typical use of the *null* driver is to make (embedded) models silent by disabling their output, either from the host application embedding the model 'mod' (Mosel C libraries):

```
XPRMsetdefstream(mod, XPRM_F_WRITE, "null:");
```

Similarly with other library interfaces, such as with the Mosel Java library:

```
mod.setDefaultStream(XPRM.F_OUTPUT, "null:");
```

Or from within Mosel, such as in the master model that controls the execution of a submodel 'submod':

```
setdefstream(submod, F_OUTPUT, "null:")
```

Or even directly in the (sub)model itself:

```
fopen(F_OUTPUT, "null:")
```

6 Exchange of information with embedded models

In this section we show several examples of the use of I/O drivers in the Mosel distribution. The full documentation of these drivers is contained in the 'Mosel Libraries Reference Manual'. When exchanging data between a Mosel model and a host application, the employed data structures need to match the representation of data in the particular host language and the choice of communication mechanisms (*e.g.*, memory blocks, streams) depends on the functionality available in this language. Each Mosel library interface therefore provides its own I/O drivers, presented as subsections to the following examples that are organized around the functionality provided by Mosel's C libraries.

6.1 `mem`: working in memory

The *mem* I/O driver makes it possible to use a block of memory as a data source. The following example shows how to use this driver to compile a Mosel model to memory (instead of creating a physical BIM file). This means that the resulting application does not generate any auxiliary file for handling the model. This feature may be of interest especially in distributed systems.

In the code printed below and in all subsequent examples the error handling is omitted for clarity's sake. The full examples are on the Xpress website.

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

int main()
{
 XPRMmodel mod;
 int result;
 char bimfile[2000];             /* Buffer to store BIM file */
 size_t bimfile_size;            /* Buffer to store actual size of BIM file */
```

```
 char bimfile_name[64];           /* File name of BIM file */

 XPRMinit();                      /* Initialize Mosel */

/* Prepare file name for compilation using 'mem' driver: */
/*   "mem:base address/size[/actual size pointer]"       */
 bimfile_size=0;
 sprintf(bimfile_name, "mem:%p/%d/%p",
         bimfile, (int)sizeof(bimfile), &bimfile_size);

                                  /* Compile model file to memory */
 XPRMcompmod(NULL, "burglar2.mos", bimfile_name, "Knapsack example");
 printf("BIM file uses %lu bytes of memory.\n", bimfile_size);

                                  /* Load the BIM file from memory */
 mod=XPRMloadmod(bimfile_name, NULL);

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

 return 0;
}
```

The only change to standard use of the Mosel library functions is the replacement of the (physical) BIM file by an extended filename in function `XPRMcompmod`. The extended filename indicates the driver we want to use (`mem`), an address where to store the BIM file, and the space reserved for it. Notice that the size of the resulting BIM file is not known before compilation. In this example we reserve roughly twice the size of the model file for the BIM file. However, there is no guarantee that this is sufficient for other models. We therefore print out the size of the generated file as a control.

We may take working in memory one step further, by including the source of the Mosel model directly in the C source code. After compilation, we obtain a single all-in-one application file that does not require or generate any auxiliary file for handling the model.

The source of the model is included in the C source file as an array of characters:

```
const char source_of_model[]=
"model Burglar\n"
"uses 'mmxprs'\n"

"declarations\n"
" WTMAX = 102                    ! Maximum weight allowed\n"
" ITEMS = {'camera', 'necklace', 'vase', 'picture', 'tv', 'video', \n"
"          'chest', 'brick'}     ! Index set for items\n"
" VALUE: array(ITEMS) of real    ! Value of items\n"
" WEIGHT: array(ITEMS) of real   ! Weight of items\n"
" take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise\n"
"end-declarations\n"

"VALUE::(['camera', 'necklace', 'vase', 'picture', 'tv', 'video',\n"
"         'chest', 'brick'])[15,100,90,60,40,15,10,1]\n"
"WEIGHT::(['camera', 'necklace', 'vase', 'picture', 'tv', 'video',\n"
"          'chest', 'brick'])[2,20,20,30,40,30,60,10]\n"

"! Objective: maximize total value\n"
"MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)\n"

"! Weight restriction\n"
"sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX\n"
"! All variables are 0/1\n"
"forall(i in ITEMS) take(i) is_binary\n"

"maximize(MaxVal)                ! Solve the problem\n"
```

```
"! Print out the solution\n"
"writeln(\"Solution:\\n Objective: \", getobjval)\n"
"forall(i in ITEMS)  writeln(' take(', i, '): ', getsol(take(i)))\n"

"end-model";
```

In the compilation command we replace the filename `burglar2.mos` by an extended filename, consisting in the driver to be used (`mem`) and the address and size of the model source.

```
 char mosfile_name[40];           /* File name of MOS file */

 sprintf(mosfile_name, "mem:%p/%d",
         source_of_model, (int)sizeof(source_of_model));

                                  /* Compile model from memory to memory */
 XPRMcompmod(NULL, mosfile_name, bimfile_name, "Knapsack example");
```

The data in this Mosel model are hardcoded directly in the model. In the following section we show how to extend this example with data input from memory, using the *raw* driver.

6.1.1 Java version of the example

The Java implementation of the model version compiling a Mosel model held in memory to a BIM file that is equally stored in memory uses objects of type `ByteBuffer` to communicate with Mosel.

```
// The source of the model as a string
 static final String source_of_model=
  "model Burglar\n"+
  "uses 'mmxprs'\n"+
...
  "end-model";


 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  ByteBuffer mosfile;                  // Buffer to store source file
  ByteBuffer bimfile;                  // Buffer to store BIM file

  mosel = new XPRM();                  // Initialize Mosel

                                       // Prepare file names for compilation:
                                       // Wrap source in a byte buffer
  mosfile=ByteBuffer.wrap(source_of_model.getBytes());
  bimfile=ByteBuffer.allocateDirect(2048);     // Create a 2K byte buffer
  mosel.bind("mybim", bimfile);        // Associate Java object
  mosel.bind("modelfile", mosfile);    // with a name in Mosel

  try
  {                                    // Compile model from memory to memory
   mosel.compile("", "java:modelfile", "java:mybim", "");
  }
  catch(XPRMCompileException e)
  {
   System.out.println(e.getMessage());
  }
  mosel.unbind("mosfile");             // Release memory
  mosfile=null;

  bimfile.limit(bimfile.position());   // Mark end of data in the buffer
  bimfile.rewind();                    // Back to the beginning
  System.out.println("BIM file uses "+bimfile.limit()+" bytes of memory.");
```

```
 mod=mosel.loadModel("java:mybim");    // Load BIM file from memory
 mosel.unbind("mybim");                // Release memory
 bimfile=null;

 mod.run();                            // Run the model
}
```

6.1.2 .NET version of the example

The .NET interface defines a combined compile+load method that keeps a temporary BIM file in memory. We can therefore simply write the following program to compile a Mosel model from and to memory.

```
// String containing the model
 const string modelSource=
    "model Burglar\n"+
    "uses 'mmxprs'\n"+
...
    "end-model";

// Main entry point for the application
 static void Main(string[] args) {
  // Initialize Mosel
  XPRM mosel = XPRM.Init();

  // Compile and load the Mosel model
  XPRMModel model = mosel.CompileAndLoad(new StringReader(modelSource));

  // Run the model
  model.Run();
 }
```

If however, we explicitly wish to create a physical BIM file we can use the separate `Compile` and `LoadModel` methods:

```
 static void Main(string[] args) {
  // Initialize Mosel
  XPRM mosel = XPRM.Init();

  // Compile the Mosel model to a physical file
  FileStream file= new FileStream("burglar2.bim", FileMode.Create, FileAccess.Write);
  mosel.Compile("", new StringReader(modelSource), file);
  file.Close();

  // Load the Mosel model
  XPRMModel model = mosel.LoadModel("burglar2.bim");

  // Run the model
  model.Run();
 }
```

6.1.3 mmjobs version of the example

An alternative form of 'embedding' a Mosel model is running a (sub)model from another Mosel model (the *master model*). The functionality for managing and coordinating multiple models is provided by the module *mmjobs* (documented in the chapter 'mmjobs´ of the 'Mosel Language Reference Manual´). The reader is referred to the Mosel whitepaper 'Multiple models and parallel solving with Mosel' for a comprehensive set of examples. With *mmjobs* we use the driver *shmem* for saving data in shared memory (memory accessible to models running in the same Mosel

instance).

```
model "Run model burglar"
 uses "mmjobs"

 declarations
  modBurg: Model                    ! Submodel
 end-declarations
                                    ! Compile the model, save bim in memory
 if compile("", "burglar2.mos", "shmem:burglar2.bim")<>0 then exit(1); end-if
 load(modBurg, "shmem:burglar2.bim") ! Load the bim file
 run(modBurg)                       ! Start model execution
 wait                               ! Wait for model termination
 dropnextevent                      ! Ignore termination event message

end-model
```

6.2 `raw`: binary data format of the host application

The *raw* I/O driver provides an implementation of the `initializations` block in binary mode: instead of translating information from/to text format, data is kept in its raw representation. The following example shows how to use this driver in combination with the *mem* driver to exchange arrays of data between a model and an application through memory. As shown in the previous section, we may work with a separate model file (this example) or alternatively include the model source directly in the model.

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

const struct
{                                   /* Initial values for array 'data': */
 const char *ind;                   /*   index name */
 double val,wght;                   /*   value and weight data entries */
} data[]={{"camera",15,2}, {"necklace",100,20}, {"vase",90,20},
          {"picture",60,30}, {"tv",40,40}, {"video",15,30},
          {"chest",10,60}, {"brick",1,10}};

double solution[8];                 /* Array for solution values */

int main()
{
 XPRMmodel mod;
 int result;
 XPRMalltypes rvalue, itemname;
 XPRMset set;
 char bimfile[2000];                /* Buffer to store BIM file */
 size_t bimfile_size;               /* Buffer to store actual size of BIM file */
 char bimfile_name[64];             /* File name of BIM file */
 char data_name[40];                /* File name of initial values for 'data' */
 char solution_name[40];            /* File name of solution values */
 char params[96];                   /* Parameter string for model execution */

 XPRMinit();                        /* Initialize Mosel */

/* Prepare file name for compilation using 'mem' driver: */
/*   "mem:base address/size[/actual size pointer]"       */
 bimfile_size=0;
 sprintf(bimfile_name, "mem:%p/%d/%p",
         bimfile, (int)sizeof(bimfile), &bimfile_size);

                                    /* Compile model file to memory */
 XPRMcompmod(NULL, "burglar2r.mos", bimfile_name, "Knapsack example");
 printf("BIM file uses %lu bytes of memory.\n", bimfile_size);
```

```
                                /* Load a BIM file from memory */
 mod=XPRMloadmod(bimfile_name, NULL);

/* Prepare file names for 'initializations' using the 'raw' driver:         */
/*   "rawoption[,...],filename"                                             */
/*   (Here, 'filename' uses the 'mem' driver, data is stored in memory)     */
/* Options for 'raw':                                                       */
/* 'slength=0': strings are represented by  pointers to null terminated     */
/*              arrays of characters (C-string) instead of fixed size arrays*/
/* 'noindex':   only array values are expected - no indices requested       */

 sprintf(data_name, "slength=0,mem:%p/%d", data, (int)sizeof(data));
 sprintf(solution_name, "noindex,mem:%p/%d", solution, (int)sizeof(solution));

                                /* Pass file names as execution param.s */
 sprintf(params, "DATA='%s',SOL='%s'", data_name, solution_name);

 XPRMrunmod(mod, &result, params);   /* Run the model */

/* Display solutions values obtained from the model */
 printf("Objective: %g\n", XPRMgetobjval(mod));
 XPRMfindident(mod, "ITEMS", &rvalue);     /* Get the model object 'ITEMS' */
 set = rvalue.set;
 for(i=0;i<8;i++)
  printf(" take(%s): %g\n", XPRMgetelsetval(set, i+1, &itemname)->string,
         solution[i]);

 return 0;
}
```

The input data in the C code above is stored and communicated to the model in *sparse format*, that is, each data item with its index tuple. For retrieving and saving the solution data this example uses *dense format*, that is, just the data items without explicitly specifying the indices. To obtain a printout with the (string) index names, we retrieve separately the set of indices from the Mosel model. Alternatively, we could also use sparse format for the solution data similarly to the input data, in which case we did not have to retrieve the index set separately.

The model file `burglar2r.mos` of this example is very similar to the model versions for the *odbc* and *diskdata* I/O drivers in Section 4. The only changes made are to the name of the I/O driver and to the strings `DATA` and `SOL` for the data source and the destination of results respectively.

```
model Burglar2r
 uses 'mmxprs'

 parameters
  IODRV = 'raw:'
  DATA=''
  SOL=''
  WTMAX = 102                    ! Maximum weight allowed
 end-parameters

 declarations
  ITEMS: set of string           ! Index set for items
  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items
  SOLTAKE: array(ITEMS) of real  ! Solution values
 end-declarations

 initialisations from IODRV
  [VALUE,WEIGHT] as DATA
 end-initialisations

 declarations
  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations
```

```
! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)               ! Solve the problem

! Solution output
 forall(i in ITEMS) SOLTAKE(i):= getsol(take(i))

 initialisations to IODRV
  SOLTAKE as SOL
 end-initialisations

end-model
```

The example above uses sparse data format, that is, every data entry comprises its index (or index tuple) and the actual value (or list of values in our case). A model version using dense data format, so communicating just the data values without the indices, is described in Chapter 13.4.1 'Exchanging data between an application and a modelDense arrays' of the 'Mosel User Guide'.

The *raw* format communicates data via fixed-size memory blocks. Greater flexibility through dynamic sizing of the data structures can be achieved by using the callback functionality provided through the driver *cb* (see Section 6.3 below and the examples of Chapters 13.4.3 and 14.1.7.3 'Exchanging data between an application and a model Dynamic data' of the 'Mosel User Guide' for C and Java examples respectively).

6.2.1 Java version of the example

With the Mosel Java libraries, we replace the C driver *raw* by the Java binary format I/O driver *jraw*. Just as in the C version, we have stored the model data in a single structure containing the string index and the two data values for 'weight' and 'value' of every item. Mosel returns the solution into a second structure that also stores index and data values.

```
// Class to store initial values for array 'data'
 public static class MyData
 {
  public String ind;                // index name
  public double val,wght;           // value and weight data entries
  MyData(String i, double v, double w)
  { ind=i; val=v; wght=w; }
 }
                       // Class to receive solution values
 public static class MySol
 {
  public String ind;                // index name
  public double val;                // solution value
 }

 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  MyData data[]={new MyData("camera",15,2), new MyData("necklace",100,20),
                 new MyData("vase",90,20), new MyData("picture",60,30),
 new MyData("tv",40,40), new MyData("video",15,30),
                 new MyData("chest",10,60), new MyData("brick",1,10)};
```

```
 MySol[] solution=new MySol[8];

 for(int i=0;i<8;i++) solution[i] = new MySol();

 mosel = new XPRM();                 // Initialize Mosel

 mosel.compile("burglar2r.mos");     // Compile & load the model
 mod = mosel.loadModel("burglar2r.bim");

                      // Associate the Java objects with names in Mosel
 mosel.bind("dt", data);
 mosel.bind("sol", solution);
                      // File names are passed through execution parameters
 mod.execParams = "DATA='dt(ind,val,wght)',SOL='sol(ind,val)',IODRV='jraw:'";

 mod.run();                          // Run the model

 if(mod.getProblemStatus()!=mod.PB_OPTIMAL)
  System.exit(1);                    // Stop if no solution found

                      // Display solution values obtained from the model
 System.out.println("Objective value: " + mod.getObjectiveValue());
 for(int i=0;i<8;i++)
  System.out.println(" take(" + solution[i].ind + "): " + solution[i].val);

 mod.reset();                        // Reset the model
}
```

A model version using dense data format is discussed in Chapter 14.1.7.1 'Exchanging data between an application and a modelDense arrays' of the 'Mosel User Guide'.

6.2.2 .NET version of the example

The .NET version of our program is again very similar to its Java version. We now use the I/O driver *dotnetraw* in the place of *jraw*.

```
// Arrays containing initialization data for the model
 static double[] vdata = new double[] {15,100,90,60,40,15,10, 1};  // VALUE
 static double[] wdata = new double[] { 2, 20,20,30,40,30,60,10};  // WEIGHT

 // Structure to store initial values for the array 'data'
 class MyData {
  public string ind;
  public double val;
  public double wght;

  public MyData(string i, double v, double w) {
   this.ind = i;
   this.val = v;
   this.wght = w;
  }
 }

// Structure to receive solution values
 class MySol {
  public string ind;
  public double val;
 }

// The initial values for the array 'data'
 private static MyData[] data = new MyData[] {
                 new MyData("camera",15,2), new MyData("necklace",100,20),
                 new MyData("vase",90,20), new MyData("picture",60,30),
 new MyData("tv",40,40), new MyData("video",15,30),
```

```
                new MyData("chest",10,60), new MyData("brick",1,10) };

// Main entry point for the application
 static void Main(string[] args) {
  // Initialize Mosel
  XPRM mosel = XPRM.Init();

  // Compile and load the Mosel model
  XPRMModel model = mosel.CompileAndLoad("burglar2r.mos");

  // Associate the .NET object with a name in Mosel
  model.Bind("dt", data);

  // Create a new array for solution data and bind that to the name 'SOL'
  MySol[] solution=new MySol[8];
  for(int i=0;i<8;i++) solution[i] = new MySol();
  mosel.Bind("sol", solution);

  // Pass data location as a parameter to the model
  model.ExecParams =
      "DATA='dt(ind,val,wght)',SOL='sol(ind,val)',IODRV='dotnetraw:'";

  // Run the model
  model.Run();

  // Print the solution
  Console.WriteLine("Objective value: {0}", model.ObjectiveValue);
  for(int i=0;i<8;i++)
   Console.Write(" take({0}): {1}",  solution[i].ind, solution[i].val);
  Console.WriteLine();
 }
```

6.2.3 mmjobs version of the example

When running a Mosel model from another model, the two models can exchange data in memory using the binary format defined by *raw*. Given the restrictions of the *raw* format (each 'file' only contains a single data array, composite Mosel data structures are not supported) we recommend however to give preference to the more flexible binary format defined by the *bin* driver that supports the full set of data structures of initializations blocks (see Section 7 below). Here is an example of a Mosel model reading and writing data to memory in binary format using *bin*.

```
model "Run model burglar IO"
 uses "mmjobs"

 declarations
  modBurg: Model                    ! Submodel
  ISet,SSet: set of string          ! Index set for data arrays
  V,W: array(ISet) of real          ! Data arrays
  Solution: array(SSet) of real     ! Solution values
 end-declarations

 V:: (["camera","necklace","vase","picture","tv","video","chest","brick"])
      [15, 100, 90, 60, 40, 15, 10,  1]
 W:: (["camera","necklace","vase","picture","tv","video","chest","brick"])
      [ 2,  20, 20, 30, 40, 30, 60, 10]

                                   ! Compile the model, save bim in memory
 if compile("burglar2r.mos")<>0 then exit(1); end-if
 load(modBurg, "burglar2r.bim")     ! Load the bim file

 setdefstream(modBurg, F_OUTPUT, "null:")   ! Disable output from submodel

! Save data in shared memory
 IODRV:="bin:shmem:burgdata"
```

```
 initializations to IODRV
  [V,W] as "Burgdata"
 end-initializations
                                ! Start model execution, setting parameters
 run(modBurg, "IODRV=’" +IODRV + "’,DATA=’Burgdata’,SOL=’Solution’")
 wait                           ! Wait for model termination
 dropnextevent                  ! Ignore termination event message

! Retrieve solution from shared memory
 initializations from IODRV
  Solution
 end-initializations

 forall(i in SSet)  writeln(" take(", i, "): ", Solution(i))

end-model
```

The relevant part of the model above look as follows with the *raw* I/O driver.

```
! Save data in shared memory
 IODRV:="raw:"
 initializations to IODRV
  [V,W] as "shmem:Burgdata"
 end-initializations
                                ! Start model execution, setting parameters
 run(modBurg, "IODRV=’" +IODRV + "’,DATA=’shmem:Burgdata’,SOL=’shmem:SolTake’")
 wait                           ! Wait for model termination
 dropnextevent                  ! Ignore termination event message

! Retrieve solution from shared memory
 initializations from IODRV
  Solution as "shmem:SolTake"
 end-initializations
```

6.3 `cb`: output to callback functions

Using the *cb* I/O driver a function is used as a file. Depending on the type of stream to manage (*i.e.* a general stream or a for an initializations block) a specific function type as to be provided. In the following example we define a callback function to receive the output stream. Our function simply prints all output lines preceeded by `Mosel:` .

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

/* Callback function to handle output */
long XPRM_RTC cbmsg(XPRMmodel model, void *info, char *buf, unsigned long size)
{
/* Note: ’model’ is NULL if the stream is used outside of an execution */
 printf("Mosel: %.*s", (int)size, buf);
 return 0;
}

int main()
{
 XPRMmodel mod;
 int result;
 char outfile_name[40];          /* File name of output stream */

 XPRMinit();                     /* Initialize Mosel */

                                 /* Prepare file name for output stream   */
                                 /* using ’cb’ driver:                    */
                                 /* "cb:function pointer[/callback data]" */
 sprintf(outfile_name, "cb:%p", cbmsg);
```

```
                                /* Set default output stream to callback */
 XPRMsetdefstream(NULL, XPRM_F_WRITE, outfile_name);

                                /* Execute = compile/load/run model file
                                           'burglar2.mos'
                                   Generates the BIM file 'burglar2.bim' */
 XPRMexecmod(NULL, "burglar2.mos", NULL, &result, NULL);

 return 0;
}
```

The output produced by this program looks as follows:

```
Mosel: Solution:
Mosel:  Objective: 280
Mosel: [('camera',1),('necklace',1),('vase',1),('picture',1),('tv',0),
('video',1),('chest',0),('brick',0)]
```

Callback functionality can also be used for data I/O in `initializations` blocks—see Chapters 13.4.3 and 14.1.7.3 (Exchanging data between an application and a model Dynamic data) of the 'Mosel User Guide' for C and Java examples respectively.

6.3.1 Java version of the example

The stream redirections functions of the Mosel Java library expect objects of the type `InputStream` or `OutputStream`. The following Java program implements the same behavior as the C program above. With Java, we replace the *cb* driver by the Java memory driver *java*.

```
// OutputStream class to handle default output
 public static class MyOut extends OutputStream
 {
  public void flush()
  { System.out.flush(); }
  public void write(byte[] b)
  {
   System.out.print("Mosel: ");
   System.out.write(b, 0, b.length);
  }
  // The following methods are not used by Mosel:
  public void write(byte[] b, int off, int len) {}
  public void write(int b) {}
  public void close() {}
 }

 public static void main(String[] args) throws Exception
 {
  XPRM mosel;
  XPRMModel mod;
  MyOut cbmsg = new MyOut();            // Define output stream as "MyOut"

  mosel = new XPRM();                   // Initialize Mosel

  mosel.bind("mycb", cbmsg);    // Associate Java object with a name in Mosel
                                // Set default output stream to cbmsg
  mosel.setDefaultStream(XPRM.F_OUTPUT|XPRM.F_LINBUF, "java:mycb");

  mosel.compile("burglar2.mos");        // Compile, load & run the model
  mod = mosel.loadModel("burglar2.bim");
  mod.run();
 }
```

6.3.2 .NET version of the example

With the Mosel .NET libraries, the stream redirection functions accept the objects `TextWriter` and `TextReader` in addition to strings with Mosel file names. We therefore have several different implementation options for our example.

The following version of our example closely matches the structure of the Java program from the previous section—it makes use of the *dotnet* I/O driver for redirecting the model output to a custom `TextWriter` object:

```
static void Main(string[] args) {
 // Initialize Mosel
 XPRM mosel = XPRM.Init();

 // Associate .NET object with a name in Mosel
 mosel.Bind("mycb", new MyOut());

 // Compile and load the Mosel model
 XPRMModel model = mosel.CompileAndLoad("burglar2.mos");

 // Redirect the model's output to our printing function 'cbmsg'
 model.SetDefaultStream(XPRMStreamType.F_OUTPUT_LINEBUF, "dotnet:mycb");

 // Run the model
 model.Run();
}
```

Alternatively, we can simply use this form of our program (without employing any I/O driver):

```
static void Main(string[] args) {
 // Initialize Mosel
 XPRM mosel = XPRM.Init();

 // Compile and load the Mosel model
 XPRMModel model = mosel.CompileAndLoad("burglar2.mos");

 // Redirect the model's output to a custom TextWriter
 MyOut modelOut = new MyOut();
 model.SetDefaultStream(XPRMStreamType.F_OUTPUT_LINEBUF, modelOut);

 // Run the model
 model.Run();
}
```

In both cases, we work with the following definition for the class `MyOut`:

```
public class MyOut: TextWriter
{
 private bool atStartOfLine = true;
 public override void Write(char b)
 {
   if (atStartOfLine) {
    Console.Write("Mosel: ");
    atStartOfLine=false;
   }
   if (b=='\n') {
    Console.WriteLine();
    atStartOfLine=true;
   }
   else if (b=='\r') {
    // ignore
   }
   else {
```

```
    Console.Write(b);
   }
  }
  public override Encoding Encoding {
   get {
    return Encoding.UTF8;
   }
  }
 }
```

6.4 `sysfd`: working with system file descriptors

With the *sysfd* driver a file descriptor provided by the operating system can be used in place of a file. File descriptors are returned by the C functions `open` or `fileno`. They also include the default input, output, and error streams (values 0, 1, and 2 respectively) that are opened automatically when a program starts. For instance, to redirect the error stream for a model `mod` to the default output we may write:

```
XPRMsetdefstream(mod, XPRM_F_ERROR, "sysfd:1");
```

6.4.1 Java version of the example

In Java, we use the driver *java* to indicate redirection to a system stream.

```
mod.setDefaultStream(XPRM.F_ERROR, "java:java.lang.System.out");
```

6.4.2 .NET version of the example

With .NET we can immediately specify an output location, such as

```
model.SetDefaultStream(XPRMStreamType.F_ERROR, Console.Out);
```

7 `bin`: using Mosel's binary format

The *bin* driver implements a structured, architecture-independent binary format that can handle cross platform files and supports the full set of data structures that may be contained in initializations blocks. It can be used for exchanging data between Mosel models or between a Mosel model and a host application, particularly in the context of distributed applications using the Mosel remote invocation library XPRD (see the program examples in Part 'V. Remote invocation of Mosel' of the 'Xpress Mosel User Guide'). The *bindrv* library that provides the binary reading and writing functionality on the library level is independent of Mosel and can be used with any C or Java program. It comes with a documentation of its own, the 'bindrv Library Reference Manual'.

Mosel code like the following expects input and output data from files in binary format.

```
initialisations from "bin:burgdatabin"
 [VALUE, WEIGHT] as "BurgData"
end-initialisations

...

initialisations to "bin:resdatabin"
 SOLTAKE as "SolTake"
```

```
 evaluation of getobjval as "Objective"
end-initialisations
```

The file binary file `burgdatabin` has the same structure as the text file `burglar.dat` that we have seen in Section 3.

```
BurgData: [(camera)    [ 15   2]
           (necklace)  [100  20]
           ...
           (brick)     [  1  10] ]
```

The following C code produces the corresponding file `burgdatabin` in *bindrv* binary format. Notice the use of controls (label, open and close a list, open and close an index) through which we represent the structure of the file.

```
#include "bindrv.h"

/**** Input data ****/
static double vdata[]={15,100,90,60,40,15,10, 1};   /* VALUE */
static double wdata[]={ 2, 20,20,30,40,30,60,10};   /* WEIGHT */
static char *ind[]={"camera", "necklace", "vase", "picture", "tv",
                    "video", "chest", "brick" };     /* Index names */
int datasize=8;

/**** Create a BinDrv data file ****/
static void writeburgbin(void)
{
 FILE *f;
 s_bindrvctx bdrv;
 int i;

 f=fopen("burgdatabin","w");
 bdrv=bindrv_newwriter((size_t (*)(const void *,size_t,size_t,void*))fwrite,f);

 bindrv_putctrl(bdrv,BINDRV_CTRL_LABEL); bindrv_putstring(bdrv,"BurgData");
 bindrv_putctrl(bdrv,BINDRV_CTRL_OPENLST);         /* [       */
 for(i=0;i<datasize;i++)
 {
  bindrv_putctrl(bdrv,BINDRV_CTRL_OPENNDX);        /*  (      */
   bindrv_putstring(bdrv,ind[i]);                  /*   index */
  bindrv_putctrl(bdrv,BINDRV_CTRL_CLOSENDX);       /*  )      */
  bindrv_putctrl(bdrv,BINDRV_CTRL_OPENLST);        /*   [     */
   bindrv_putreal(bdrv,vdata[i]);                  /*    val1 */
   bindrv_putreal(bdrv,wdata[i]);                  /*    val2 */
  bindrv_putctrl(bdrv,BINDRV_CTRL_CLOSELST);       /*   ]     */
 }
 bindrv_putctrl(bdrv,BINDRV_CTRL_CLOSELST);        /* ]       */

 bindrv_delete(bdrv);
 fclose(f);
}
```

The structure of the binary result file `resdatabin` corresponds to this text file:

```
'SolTake': [('camera') 1 ('necklace') 1 ('vase') 1 ('picture') 1 ('tv') 0
 ('video') 1 ('chest') 0 ('brick') 0]
'Objective': 280
```

The binary file can be read and displayed with the following C code—the routines `bindrv_get*` follow the same logic as the `bindrv_put*` routines we have seen in the writing function, decoding the file structure via controls for labels, square and round brackets (for lists and indices respectively). For clarity's sake, we have implemented only a limited amount of checks whether

the detected structure and data types correspond to the expected format.

```
static void readburgbin(void)
{
 FILE *f;
 s_bindrvctx bdrv;
 union { int i; double r; char b; char *str; BINDRV_LONG l;} val;

 f=fopen("resdatabin","r");
 bdrv=bindrv_newreader((size_t (*)(void *,size_t,size_t,void*))fread,f);

 bindrv_getctrl(bdrv,&(val.i));
 if(val.i==BINDRV_CTRL_LABEL)
 {
  while(bindrv_nexttoken(bdrv)>=0)
  {
   bindrv_getstring(bdrv,&(val.str));                    /* label:  */
   if(strcmp(val.str,"SolTake")==0)
   {
    free(val.str);
    printf("Solution values:\n");
    bindrv_getctrl(bdrv,&(val.i));                       /* [       */
    while(bindrv_nexttoken(bdrv)>=0)
    {
     bindrv_getctrl(bdrv,&(val.i));
     if(val.i== BINDRV_CTRL_LABEL)  break;
     switch(val.i)
     {
      case BINDRV_CTRL_OPENNDX:                           /*  (      */
       printf(" take(");
       bindrv_getstring(bdrv,&(val.str));                 /*   index */
       printf("%s",val.str);
       bindrv_getctrl(bdrv,&(val.i));                     /*  )      */
       printf(")=");
       bindrv_getreal(bdrv,&(val.r));                     /*  value  */
       printf(" %g\n",val.r);
       break;
      case BINDRV_CTRL_CLOSELST:                          /* ]       */
       printf("\n");
       break;
      default:
       printf("Unexpected token %d\n", val.i); exit(1);
     }
    }
   }
   else if(strcmp(val.str,"Objective")==0)
   {
    free(val.str);
    bindrv_getreal(bdrv,&(val.r));
    printf("Objective value = %g\n", val.r);
   }
   else
   {
    printf("Unexpected label '%s'\n", val.str);
    free(val.str);
    exit(1);
   }
  }
 }
 else
 {
  printf("Unexpected token '%d'\n", val.i);
  exit(1);
 }

 bindrv_delete(bdrv);
 fclose(f);
```

```
}
```

7.1 Java version of the example

The same binary files can be written and read with the Java version of the Bindrv library. The code producing the binary file `burgdatabin` looks as follows with Java—some of the calls to methods `bdrv.put*` have been combined for improved readability of our Java code (this does not impact the generated output).

```
/**** Input data ****/
 static final double[] vdata={15,100,90,60,40,15,10, 1};   // VALUE
 static final double[] wdata={ 2, 20,20,30,40,30,60,10};   // WEIGHT
 static final String[] ind={"camera", "necklace", "vase", "picture", "tv",
                  "video", "chest", "brick" };            // Index names
 static final int datasize=8;

/**** Create a BinDrv data file ****/
 static void writeBurgBin() throws IOException
 {
  BinDrvWriter bdrv;
  FileOutputStream f;

  f=new FileOutputStream("burgdatabin");
  bdrv=new BinDrvWriter(f);

  bdrv.putControl(bdrv.CTRL_LABEL).put("BurgData");      // label:
  bdrv.putControl(bdrv.CTRL_OPENLST);                    // [
  for(int i=0;i<datasize;i++)
  {                                                      //  (index)
   bdrv.putControl(bdrv.CTRL_OPENNDX).put(ind[i]).putControl(bdrv.CTRL_CLOSENDX);
   bdrv.putControl(bdrv.CTRL_OPENLST).put(vdata[i]).put(wdata[i]);
   bdrv.putControl(bdrv.CTRL_CLOSELST);                  //   [val1 val2]
  }
  bdrv.putControl(bdrv.CTRL_CLOSELST);                   // ]
  f.close();
 }
```

The Java code for reading the binary results file `resdatabin` has the same structure as the C version that we have seen in the previous section. Again, we have implemented only a limited number of tests whether the data read corresponds to the expected format.

```
 static void readBurgBin() throws IOException
 {
  BinDrvReader bdrv;
  FileInputStream f;
  int c;

  f=new FileInputStream("resdatabin");
  bdrv=new BinDrvReader(f);

  if(bdrv.getControl()==BinDrvReader.CTRL_LABEL)
  {
   while(bdrv.nextToken()>=0)
   {
    String s=bdrv.getString();                           // label:
    if(s.equals("SolTake"))
    {
     System.out.println("Solution values:");
     bdrv.getControl();
     while(bdrv.nextToken()>=0)
     {
      c=bdrv.getControl();                               // [
      if(c==BinDrvReader.CTRL_LABEL) break;
```

```
      switch(c)
      {
       case BinDrvReader.CTRL_OPENNDX:                    //  (
          System.out.print(" take("+bdrv.getString()+")="); // index
          bdrv.getControl();                              //  )
          System.out.println(bdrv.getReal());             //   value
          break;
       case BinDrvReader.CTRL_CLOSELST:                   // ]
          System.out.println(""); break;
       default:
          System.out.println("Unexpected Control");
          System.exit(0);
      }
     }
    }
    else if(s.equals("Objective"))
      System.out.println("Objective value = "+bdrv.getReal());
    else
    {
     System.out.println("Unexpected label "+s);
     System.exit(0);
    }
   }
  }
  else
  {
   System.out.println("Unexpected token");
   System.exit(1);
  }

  f.close();
 }
```

7.2 mmjobs version of the example

A model with an *mmjobs* version of the application program has already been shown in Section 6.2.3. From multiple models running on the same Mosel instance, we now move one step further, namely, running the submodel on a remote Mosel instance. Before compiling the submodel, we connect to another Mosel instance. This instance is used for compiling to model, the source of which is located on our local machine running the master model. Remote access to a file is achieved via the *rmt* driver (equally defined by module *mmjobs*). In our example, data is exchanged via shared memory (*shmem*) accessed from a remote Mosel instance (*rmt*, optionally followed by a node number in square brackets) using binary format (*bin*), resulting in extended file names that combine several driver prefixes, for example, `bin:rmt:[-1]shmem:burgdata` to access binary data held in shared memory of the parent node.

```
model "Run model burglar remotely IO"
 uses "mmjobs", "mmsystem"

 declarations
  modBurg: Model                    ! Submodel
  moselInst: Mosel                  ! Mosel instance
  ISet,SSet: set of string          ! Index set for data arrays
  V,W: array(ISet) of real ! Data arrays
  SolTake: array(SSet) of real      ! Solution values
 end-declarations

 V:: (["camera","necklace","vase","picture","tv","video","chest","brick"])
      [15, 100, 90, 60, 40, 15, 10,  1]
 W:: (["camera","necklace","vase","picture","tv","video","chest","brick"])
      [ 2,  20, 20, 30, 40, 30, 60, 10]

!!! Select the (remote) machines to be used:
```

```
!!! Use names, IP addresses, or empty string for the node running this model
 MOSINST:= ""
                                    ! Connect to a remote instance
 if connect(moselInst, MOSINST)<>0 then exit(1); end-if
                                    ! Compile the model remotely
 if compile(moselInst, "", "rmt:"+expandpath("burglar2m.mos"),
           "shmem:burglar.bim")<>0 then
  exit(2); end-if
 load(moselInst, modBurg, "shmem:burglar.bim") ! Load the bim file
 fdelete("rmt:["+getnode(moselInst)+"]shmem:burglar.bim")
                                    ! bim file is no longer needed

 setdefstream(modBurg, F_OUTPUT, "null:")      ! Disable output from submodel

! Save data in shared memory on local host
 initializations to "bin:shmem:burgdata"
  V as "VALUE"
  W as "WEIGHT"
 end-initializations
                                    ! Start model execution, setting parameters
 run(modBurg,
     "DATA='bin:rmt:[-1]shmem:burgdata',SOL='bin:rmt:[-1]shmem:burgsol'")
 wait                               ! Wait for model termination
 dropnextevent                      ! Ignore termination event message

! Retrieve solution from shared memory
 initializations from "bin:shmem:burgsol"
  SolTake
 end-initializations

 forall(i in SSet)  writeln(" take(", i, "): ", SolTake(i))

end-model
```

8 User-defined drivers

The contents of this section is quite advanced material and we recommend its reading only to those who are interested in the Mosel Native Interface (NI).

In this section we discuss two examples of user-defined drivers, performing compression and code generation respectively. New drivers are defined via the NI, in the form of *services*. For every driver some or all of a list of standard I/O functions may be defined, depending on its purpose.

The full source code of both examples is provided with the examples of the Mosel distribution. This compression module *zlib* is also included in compiled form among the DSOs of the Mosel distribution. For more detail on the implementation of new I/O drivers see the 'Mosel NI User Guide' and 'Mosel NI Reference Manual'.

8.1 Example: compression

With the *gzip* I/O driver for file compression implemented by the example module *zlib*, compressed files may be used for Mosel input and output. For example, we may *generate and run a compressed BIM file* with the following command:

```
mosel comp mymodel.mos -c "example of compression" -o zlib.gzip:mymodel.gz
mosel run zlib.gzip:mymodel.gz
```

To *use compressed data files* in a model we may write:

```
initializations from "zlib.gzip:burglar.dat.gz"
```

Another use of this driver may be to generate *compressed matrix files*:

```
exportprob("zlib.gzip:prob.mat.gz")
```

The I/O driver *gzip* implements an interface to the gzip compression format of the *ZLIB* library. The complete example module *zlib* also implements a second driver, *compress*, for a different compression format defined by this library. The driver *gzip* works with physical files, *compress* is a more general driver that works with streams. The *ZLIB* library is required to execute this example: it can be found at http://www.zlib.org.

8.1.1 Implementation

A C program defining new I/O drivers has the following components

- **Interface structures:** the DSO interface table, the table of services, the table of drivers, and a table of functions per driver.
- **Module initialization function**
- **Implementation of driver access functions**: opening and closing streams, standard reading and writing, special reading and writing for `initializations` blocks, error handling, deleting and moving files.

A driver must implement at least the stream open functionality and one of the standard or special reading or writing functions; everything else is optional.

The definition of the *interface structures* of module *zlib* is the following. The structure of all tables and the function prototypes are defined by the Mosel NI.

```
                                /* Functions of the 'gzip' driver */
static void *gzip_open(XPRMcontext ctx, int *mode, const char *fname);
static int gzip_close(XPRMcontext ctx, gzFile gzf, int mode);
static long gzip_read(XPRMcontext ctx, gzFile gzf, void *buffer,
                                                    unsigned long size);
static long gzip_write(XPRMcontext ctx, gzFile gzf, void *buffer,
                                                    unsigned long size);
static XPRMiofcttab iodrv_gzip[]=
        {
         {XPRM_IOCTRL_OPEN, gzip_open},
         {XPRM_IOCTRL_CLOSE, gzip_close},
         {XPRM_IOCTRL_READ, gzip_read},
         {XPRM_IOCTRL_WRITE, gzip_write},
         {XPRM_IOCTRL_INFO, "filename"},
         {0,NULL}
        };
                                /* Drivers of the zlib module */
static XPRMiodrvtab iodrv_zlib[]=
        {
         {"gzip", iodrv_gzip},
         {NULL,NULL}
        };
                                /* Table of services: only IO drivers */
static XPRMdsoserv tabserv[]=
        {
         {XPRM_SRV_IODRVS, iodrv_zlib}
        };
                                /* DSO interface: only services */
static XPRMdsointer dsointer=
        {
         0, NULL,
         0, NULL,
```

```
         0, NULL,
         sizeof(tabserv)/sizeof(mm_dsoserv), tabserv
        };

static XPRMnifct mm;            /* For storing Mosel NI function table */
```

The previous code extract is best read from bottom to top. It declares an object of type `XPRMnifct` to store the NI function table (to be retrieved during the initialization of the module). The main DSO interface table `dsointer` lists the functionality provided by this module: it only defines services. The table of services, `tabserv`, indicates which services are defined, namely new I/O drivers. The list of I/O drivers is given in the table `iodrv_zlib`. The functionality implemented by the I/O driver *gzip* is listed in the table `iodrv_gzip`. The function bodies of `gzip_open`, `gzip_close`, *etc.*, are printed below. The string set with `INFO` will be used by the 'help' command of Mosel to indicate the usage of this driver (it must be used with the name of a physical file, not with extended filenames).

The *module initialization function* performs the standard initialization of Mosel modules: retrieving the Mosel NI function table and the NI version number and returning the module DSO interface table to Mosel. In addition, we check the version of the ZLIB library.

```
DSO_INIT zlib_init(XPRMnifct nifct, int *interver, int *libver,
                   XPRMdsointer **interf)
{
 const char *zlv;

 mm=nifct;                      /* Save the table of Mosel NI functions */
 *interver=XPRM_NIVERS;         /* The interface version we are using */

                                /* We check the ZLIB library version here */
 zlv=zlibVersion();
 if((zlv==NULL)||(zlv[0]!=ZLIB_VERSION[0]))
 {
  mm->dispmsg(NULL,"ZLIB: wrong version (expect %s got %s).\n",ZLIB_VERSION,
                                                        (zlv!=NULL)?zlv:"0");
  return 1;
 }
 else
 {
  *libver=XPRM_MKVER(0,0,1);    /* The version of the module: 0.0.1 */
  *interf=&dsointer;            /* Our interface */
  return 0;
 }
}
```

Below are printed the *functions* implementing the *gzip* driver. Besides the obligatory 'open' function the *gzip* driver implements the functions 'closing a file', 'reading' (= uncompress), and 'writing' (= compress).

```
/**** Open a gzip-file for (de)compression ****/
static void *gzip_open(XPRMcontext ctx, int *mode, const char *fname)
{
 char cmode[16];
 int cml;

 cml=2;
 cmode[0]=((*mode)&XPRM_F_WRITE)?'w':'r';
 cmode[1]=((*mode)&XPRM_F_BINARY)?'b':'t';
 if((*mode)&XPRM_F_APPEND) cmode[cml++]='a';
 cmode[cml]='\0';
 return gzopen(fname,cmode);
}
```

```
/**** Close the gzip-file ****/
static int gzip_close(XPRMcontext ctx, gzFile gzf, int mode)
{ return gzclose(gzf); }

/**** Uncompress a block of data ****/
static long gzip_read(XPRMcontext ctx, gzFile gzf, void *buffer,
                                                   unsigned long size)
{ return gzread(gzf, buffer, size); }

/**** Compress a block of data ****/
static long gzip_write(XPRMcontext ctx, gzFile gzf, void *buffer,
                                                   unsigned long size)
{ return gzwrite(gzf, buffer, size); }
```

As mentioned earlier, the full *zlib* example module in the module examples of the Mosel distribution defines a second compression driver, *compress*, in a very similar way to what has been shown here for the *gzip* driver.

8.2 Example: code generation

In this section we present a driver that includes the output of Mosel into a C program. It will typically be used for generating a C program that includes the BIM file of a given model, for example using a command like

```
mosel comp mymodel.mos -o export.toC:mymodel.c
```

As with all Mosel I/O drivers, the *toC* driver may be combined with other drivers. For instance, we may use the *compress* driver of the *zlib* module to obtain a compressed BIM file within the generated C program:

```
mosel comp mymodel.mos -o zlib.compress:export.toC:mymodel.c
```

Note: The *deploy* module that comes as a part of the standard Mosel distribution has been designed for the specific task of generating a C program that includes the BIM file of a given model; the command

```
mosel comp mymodel.mos -o deploy.csrc:runmymodel.c
```

has the same purpose as the use of 'export.toC' shown above. However, the main interest of the *deploy* module is the possibility to generate immediately a standalone executable to run a model:

```
mosel comp mymodel.mos -o deploy.exe:runmymodel
```

8.2.1 Implementation

The module *export* that defines the *toC* I/O driver has the following components (common to all I/O driver modules):

- Interface structures
- Module initialization function
- Implementation of driver access functions

The file `export.c` implementing the module *export* with the I/O driver *toC* starts with the definition of the C code template, divided into three parts:

```
static char prg_part1[]=
"#include <stdio.h>\n"
"#include \"xprm_rt.h\"\n\n"

"static unsigned int bimfile[]={";

static char prg_part2[]=
"0};\n\n"

"int main(int argc,char *argv[])\n"
"{\n"
" char modname[40];\n"
" XPRMmodel mod;\n"
" int rts;\n\n"

" rts=XPRMinit();\n"
" if((rts!=0)&&(rts!=32))\n"
" {\n"
"  char msg[512];\n\n"

"  XPRMgetlicerrmsg(msg,512);\n"
"  fprintf(stderr,\"%s\",msg);\n"
"  return 1;\n"
" }\n\n"

" sprintf(modname,\"mem:%p/%u\", bimfile,";

static char prg_part3[]=
");\n"
" if((mod=XPRMloadmod(modname,NULL))==NULL)\n"
"  return 2;\n"
" if(XPRMrunmod(mod,&rts,NULL))\n"
"  return 3;\n"
" else\n"
"  return rts;\n"
"}\n";
```

The generated output will be inserted in between parts 1 and 2, its size will be inserted in between parts 2 and 3.

The definition of the NI *interface structures* follows the same scheme like what we have seen in the previous example: the table of driver functions, the table of drivers, the table of services, the main DSO interface table, and an address for storing the NI function table. The *toC* driver implements ‘open’, ‘close’, and ‘write’ functionality and works with generalized files.

```
static void *toc_open(XPRMcontext ctx, int *mode, const char *fname);
static int toc_close(XPRMcontext ctx, s_tocdata *td, int mode);
static long toc_write(XPRMcontext ctx, s_tocdata *td, char *buffer,
                                                      unsigned long size);
static XPRMiofcttab iodrv_toc[]=
        {
         {XPRM_IOCTRL_OPEN, (void *)toc_open},
         {XPRM_IOCTRL_CLOSE, (void *)toc_close},
         {XPRM_IOCTRL_WRITE, (void *)toc_write},
         {XPRM_IOCTRL_INFO, "extended_filename"},
         {0,NULL}
        };
                                /* Drivers of module ‘export’: toC */
static XPRMiodrvtab iodrv_export[]=
        {
         {"toC", iodrv_toc},
         {NULL,NULL}
        };
                                /* Table of services: only I/O drivers */
static XPRMdsoserv tabserv[]=
```

```
        {
         {XPRM_SRV_IODRVS, iodrv_export}
        };
                                /* DSO interface: only services */
static XPRMdsointer dsointer=
        {
         0,NULL,
         0,NULL,
         0,NULL,
         sizeof(tabserv)/sizeof(mm_dsoserv),tabserv
        };

static XPRMnifct mm;            /* For storing Mosel NI function table */
```

This module's *initialization function* performs the standard module initialization procedure where the module's interface structure is passed on to Mosel and the NI function table is saved for use by this module.

```
DSO_INIT export_init(XPRMnifct nifct, int *interver, int *libver,
                     XPRMdsointer **interf)
{
 mm=nifct;                      /* Save the table of Mosel NI functions */
 *interver=XPRM_NIVERS;         /* The interface version we are using */
 *libver=XPRM_MKVER(0,0,1);     /* The version of the module: 0.0.1 */
 *interf=&dsointer;             /* Our interface */
 return 0;
}
```

The *functions* implemented by this I/O driver are restricted to opening and closing a file and writing to a file. The 'open' and 'close' functions print the predefined part of the generated C file. The function for writing the binary part of the output (`toc_write`) is somewhat tricky since it needs to make sure that all bytes are aligned correctly. This function is called with a buffer as parameter. For every four bytes it prints the corresponding integer. Since the buffer size is not necessarily a multiple of 4 there may be a few bytes left over at the end of the buffer which must be carried over to the next call to this function or to the 'close' function. Below we print the 'open' and 'close' functions. The definition of the 'write' function is left out here, the interested reader may be reminded that the complete module source is a part of the Mosel module examples.

```
static void *toc_open(XPRMcontext ctx, int *mode, const char *fname)
{
 s_tocdata *td;
                                        /* First time: open actual file */
 if(mm->fopen(ctx,*mode,fname)<0)
 {
  *mode|=XPRM_F_SILENT;                 /* Error message already displayed */
                                        /* switch so silent mode */
  return NULL;
 }
 else
 {
  td=(s_tocdata *)malloc(sizeof(s_tocdata));
  td->total=0;
  td->nb_remain=0;
  td->nb_printed=0;
  mm->printf(ctx, "%s\n", prg_part1);  /* Display the beginning of program */
  return td;
 }
}

static int toc_close(XPRMcontext ctx, s_tocdata *td, int mode)
{
```

```
 if(td->nb_remain>0)                  /* Send bytes to be written */
 {
  td->total+=td->nb_remain;
  for(;td->nb_remain<4;td->nb_remain++)
   td->map.c[td->nb_remain]=0;
  mm->printf(ctx, "%#x, ", td->map.i);
 }                                    /* and complete the program */
 mm->printf(ctx, "%s%u%s", prg_part2, td->total, prg_part3);
 return mm->fclose(ctx,mode);
}
```

These two functions work with the following structure for storing information concerning the printed output. Function 'open' initializes the structure (`td`), function 'write' copies the remaining bytes from every buffer into this structure and sets the counters, and function 'close' checks whether there is anything left to be printed before completing and closing the output file.

```
typedef struct
        {
         unsigned int total;      /* Total number of bytes written so far */
         union
         {
          char c[4];
          unsigned int i;
         } map;                   /* Mapping between 4 chars and an integer */
         int nb_remain;           /* Number of bytes not yet written */
         int nb_printed;          /* Number of numbers written on the line */
        } s_tocdata;
```

9 Summary

The concept of I/O drivers makes it possible to work with very different notions of 'file' in Mosel models and in the Mosel libraries. With only minimal changes to his models the user may switch between data sources of different formats. The interaction and exchange of data between a model and the application executing it can be made more immediate and as a consequence, more efficient. It is also possible to avoid the creation of physical intermediate files by performing all operations in memory. The latter may be useful, for instance, in distributed applications.