User guide

FICO® Xpress Optimization

Xpress Mosel Native Interface

User guide

Release 4.8

Last update 18 October, 2017

FICO® Xpress Mosel

Deliverable Version: A

Last Revised: 18 October, 2017

Version 4.8

Contents

Introduction

The Mosel language is extensible by the means of *modules*. A module may define

- constants
- subroutines
- types
- operators for the types defined by the module
- I/O drivers
- control parameters

Constants that are used by several Mosel programs could be defined by a module; a module may also publish constants that are to be used in combination with its types or subroutines.

Subroutines are probably the most common use of modules. These may be entirely new functions or procedures, or overload existing subroutines of Mosel.

Defining new *types* requires a little more work, but as a result the user defined types will be no different from Mosel's own types (like integer or mpvar). So user defined types can be used in complex data structures (arrays, sets), read from file in `initializations` sections, appear as parameters of subroutines, or have operators applied to them.

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

Control parameters make little sense on their own. They may be used for directing the behavior of subroutines defined by a module (*e.g.* algorithmic settings) or obtaining status information from a module. The values of control parameters may be changed from within a Mosel program.

Depending on the purpose of the module, it needs to provide one or several of the following to Mosel

- a list of constants
- a list of subroutines
- a list of types
- a list of services

Services are functions that Mosel calls at predefined places to perform tasks that may be characterized as 'administration' of the module: the definition of types makes a reset functionality necessary; control parameters are retrieved and enumerated through service functions; other service functions may be activated during the checking of the version number and when Mosel unloads the module. I/O drivers are also defined as services. A module that only defines constants or subroutines may not require any specific services.

Mosel expects the required information to be formatted correctly. In the following pages we shall see a few examples how this is to be done. The first example, in Chapter 1, shows how different types of constants are defined in a module. The following chapter lists and comments the complete code of a module that implements a single subroutine. Chapters 3 and 5 give examples of the implementation of new types. In Chapter 3 this is a structure grouping data items of various types and in Chapter 5 a new numerical type is defined. Chapter 4 adds the definition of parameters to the module from Chapter 3.
A specific set of NI functions and data structures are dedicated to the generation and handling of the matrix representation for LP/MIP solvers, Chapter 6 documents an example implementation for basic solver access functionality for Xpress Optimizer.
If the Mosel program that uses a module is compiled and executed from a C program, then the definition of the module can be included directly in this C program. Chapter 7 gives an example of such a *static module*.

Prerequisites

To be able to write your own modules you have to be very familiar with the way Mosel works, specifically the Mosel libraries. The implementation of a module (especially for defining new types) requires a fair amount of programming and a good experience in C programming is recommended.

Standard elements of a module

The following may serve as a check list for writing modules and a quick reference as to where to find the corresponding examples and documentation in this user guide:

- Module initialization (always required):
  - Mosel Native Interface header function: 1.2, 2.3, 7.3
  - main interface structure
    examples: 1.2.2, 2.2.2, 2.3, 3.2.4, 6.2.6, 7.3;
  - structure with the list of NI functions
    example: 2.2.3, 2.3, 7.3
  - initialization function
    examples: 1.2.3, 2.2.3, 2.3, 6.2.7, 7.2.2; documentation: A.1
  - module context
    examples: 3.2.5, 4.2.3, 5.4, 6.2.5
- Definition of constants:
  - entry in the list of constants
    examples: 1.2.1, 6.2.2; documentation: A.2.1
- Definition of subroutines:

  - entry in the list of subroutines
    examples: 2.2.1, 3.2.2, 4.2.1, 5.2.1, 6.2.1, 7.2.1; documentation: A.2.2
  - implementation
    examples: 2.3, 3.5, 4.4, 7.3, 6.3.1, 6.4.2, 6.5.1

- Definition of types:
  - entry in the list of types
    example: 3.2.1, 5.2.2, 6.2.3; documentation: A.2.3
  - entry in the list of services
    example: 3.2.3; documentation: A.2.4
  - implementation of service `XPRM_SRV_RESET`: 3.4, 5.4, 6.3.2
  - implementation of type-related functions
    documentation: A.2.3
    required function `create`: 3.3, 5.4, 6.3.3
    optional functions `delete`, `tostr`, `fromstr`, `copy`: 3.3, 5.4, 6.3.3
  - operators (entries in the list of subroutines)
    examples: 3.2, 5.2.1; documentation: A.2.2.1
  - implementation of operators
    examples: 3.5, 5.3

- Definition of I/O drivers:
  - for examples see the Xpress Whitepaper *'Generalized file handling in Mosel'*

- Definition of control parameters:
  - required parameter information (no predefined structure)
    examples: 4.2.3, 6.2.2; documentation: A.2.5
  - entries in the list of services
    examples: 4.2.2, 6.2.4; documentation: A.2.4
  - implementation of service functions
    required service `XPRM_SRV_PARAM`: 4.3, 6.3.2
    optional service `XPRM_SRV_PARLST`: 4.3, 6.3.2
  - entries in the list of subroutines
    examples: 4.2.1, 6.2.1; documentation: A.2.2
  - implementation of the subroutines `setparam` and `getparam`: 4.4, 6.3.1

- Definition of a MIP solver interface:
  - MIP solver interface structure
    example: 6.2.5
  - implementation of matrix handling functions
    examples: 6.3.1.1, 6.5

Creating a DSO

From the operating system point of view, a module is a dynamic library (Dynamic Shared Object, DSO). The name of this DSO is the name of the module with the file extension `.dso`. For instance, assuming we have written a file `test.c` to implement the module `testmodule`, the DSO will be called `testmodule.dso`. To build this DSO, under Linux the following compilation command should be used:

```
gcc -shared -D_REENTRANT -I${MOSEL}/include test.c -o testmodule.dso
```

Similarly for Unix (Sun Solaris):

```
cc -G -D_REENTRANT -I${MOSEL}/include test.c -o testmodule.dso
```

The corresponding command under Windows:

```
cl /MD /LD /Fetestmodule.dso /I%MOSEL%\include test.c
```

Example makefiles are provided with the module examples in the Mosel distribution.

Mosel looks for the DSOs in the directory `dso` under the directory that one of the environment variables `MOSEL`, `XPRESSDIR`, or `XPRESS` point to. If user-written DSOs are placed in a different directory, the environment variable `MOSEL_DSO` needs to be set to their location(s). The `MOSEL_DSO` is expected to be a list of paths conforming to the operating system conventions.

Modules vs. packages

Release 2.0 of Mosel introduced the possibility to write libraries for Mosel directly in the Mosel language, such a library is called a *package*. Packages are used from a Mosel model in exactly the same way as modules, namely by specifying their name in a `uses` statement. However, from the implementation and functionality points of view the two ways of writing Mosel libraries are not the same and the choice between packages and modules depends largely on the contents and intended use of the library. In some cases it may be convenient to split the implementation of a library into two parts, one as a module and the other as a package. If a module and a package on the specified DSO path have the same name, the package will be loaded by Mosel.

The following list summarizes the main differences between packages and modules.

- **Definition**
  - *Package*
    - library written in the Mosel language
  - *Module*
    - dynamic library written in C that obeys the conventions of the Mosel Native Interface
- **Functionality**
  - *Package*
    - define
      - symbols
      - subroutines
      - types
  - *Module*

        * extend the Mosel language with
            - constant symbols
            - subroutines
            - operators
            - types
            - control parameters
            - IO drivers

- **Efficiency**
    - *Package*
        * like standard Mosel models
    - *Module*
        * faster execution speed
        * higher development effort

- **Use**
    - *Package*
        * making parts of Mosel models re-usable
        * deployment of Mosel code whilst protecting your intellectual property
    - *Module*
        * connection to external software
        * time-critical tasks
        * definition of new I/O drivers and operators for the Mosel language

With every module example in this manual we shall discuss the possibilities of implementing similar functionality as a package. For a detailed introduction to writing packages the reader is refered to the chapter 'Packages' of the *Mosel User Guide*.

CHAPTER 1

Defining constants

Several models might share a set of constants (such as mathematical constants or text strings to obtain nicely formatted output). Defining these constants in a module that is loaded by every model makes a repetetion of the definitions in every single model unnecessary.

1.1 Example

Below we show how to define constants of different types (integer, real, string, Boolean). Once this module with the name `myconstants` is completed, we can write a simple model to output the constants:

```
model "test myconstants module"
 uses "myconstants"

 writeln(MYCST_LINE)
 writeln("BigM value: ", MYCST_BIGM, ", tolerance value: ", MYCST_TOL)
 writeln("Boolean flags: ", MYCST_FLAG, " ", MYCST_NOFLAG)
 writeln(MYCST_LINE)

end-model
```

The result that we expect to see printed is the following:

```
----
BigM value: 10000, tolerance value: 1e-05
Boolean flags: true false
----
```

Without the need to write such a test program, we could use the Mosel command

```
examine myconstants
```

which will list all constants and (if there were any) subroutines, types and parameters, defined by the module `myconstants`.

To prevent name clashes between constants that are provided by different modules, a good habit to get into is to use prefixes (e.g. based on the module name) in the names of constants, as is done in the following example.

1.2 Structures for passing information

A module that merely defines constants does not require any specific information to be passed from Mosel into the module. For the information flow from the module to Mosel, that is to make

the constants defined in the module known to Mosel, certain predefined structures must be used. These structures are defined in the header file `xprm_ni.h` which must be included by every module source file (no other Mosel header files are required):

```
#include "xprm_ni.h"
```

1.2.1 List of constants

The list of the constants and their definitions must be contained in a structure of type `XPRMdsoconst`:

```
static const double tol=0.00001;
static XPRMdsoconst tabconst[]=
   {
    XPRM_CST_INT("MYCST_BIGM", 10000),        /* A large integer value */
    XPRM_CST_REAL("MYCST_TOL", tol),          /* A tolerance value */
    XPRM_CST_STRING("MYCST_LINE",             /* String constant */
    "----"),
    XPRM_CST_BOOL("MYCST_FLAG", XPRM_TRUE),     /* Constant with value true */
    XPRM_CST_BOOL("MYCST_NOFLAG", XPRM_FALSE)   /* Constant with value false */
   };
```

In this list, the type of a constant is indicated by the macro name `XPRM_CST_`*type*. The example shows all possible types: integer, real, string, and Boolean. The first parameter of the macro is the name of the constant (in a Mosel program), the second its value. Note that double (Mosel's real) constants cannot be defined immediately in this structure, their value must be given through a C variable of type `static const double`.

1.2.2 Interface structure

The list of constants is then included in the *interface structure*. The interface structure takes the lists of constants, subroutines, types, and services (in this order) in the form of pairs *size, list* (every list is preceded by its size):

```
static XPRMdsointer dsointer=
    {
     sizeof(tabconst)/sizeof(XPRMdsoconst), tabconst,
     0, NULL,
     0, NULL,
     0, NULL
    };
```

1.2.3 Initialization function

The main exchange of information between the new module and Mosel takes place in the module *initialization function*. The format and the name of this function are fixed by Mosel:

```
DSO_INIT myconstants_init(XPRMnifct nifct, int *interver,int *libver,
                          XPRMdsointer **interf)
{
 *interver=XPRM_NIVERS;     /* Mosel NI version */
 *libver=XPRM_MKVER(0,0,1); /* Module version */
 *interf=&dsointer;         /* Pass info about module contents to Mosel */
 return 0;
}
```

The function name serves to identify this function as the one that initializes the module. It must consist of the module name followed by `_init`. With the first function parameter, Mosel passes the list of its Native Interface (NI) functions into the module (not used by this module). These

functions correspond largely to the functions of the Mosel Run Time Library, with some additional functions for modifying the model data. The remaining parameters must be filled by the module: the current Mosel Native Interface version, the version number of the module and the interface structure with all the items that are to be made known to Mosel. We set the module version number to 0.0.1. If a model file is compiled into a binary model file with this version of the module, the binary model file can be run with any version 0.0.*n* of the module, where $n \geq 1$.

1.3 Complete module example

Below follows the complete listing of the program that implements the *myconstants* module.

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

static const double tol=0.00001;

/* List of constants */
static XPRMdsoconst tabconst[]=
    {
     XPRM_CST_INT("MYCST_BIGM", 10000),   /* A large integer value */
     XPRM_CST_REAL("MYCST_TOL", tol),     /* A tolerance value */
     XPRM_CST_STRING("MYCST_LINE",        /* String constant */
     "----"),
     XPRM_CST_BOOL("MYCST_FLAG", XPRM_TRUE),     /* Constant with value true */
     XPRM_CST_BOOL("MYCST_NOFLAG", XPRM_FALSE)   /* Constant with value false */
    };

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

/* Module initialization function */
DSO_INIT myconstants_init(XPRMnifct nifct, int *interver,int *libver,
  XPRMdsointer **interf)
{
 *interver=XPRM_NIVERS;     /* Mosel NI version */
 *libver=XPRM_MKVER(0,0,1); /* Module version */
 *interf=&dsointer;         /* Pass info about module contents to Mosel */

 return 0;
}
```

1.4 Module vs. package

Identical functionality and behavior to what is provided by our module `myconstants` may be obtained from a package. The implementation of package `myconstants` (see *Mosel User Guide*, chapter 'Packages' for further explanation) takes less than 10 lines of Mosel code, making our C implementation appear unnecessarily complicated for the definition of a few constants:

```
package myconstants

 public declarations
  MYCST_BIGM = 10000          ! A large integer value
  MYCST_TOL = 0.00001         ! A tolerance value
  MYCST_LINE = "----"         ! String constant
  MYCST_FLAG = true           ! Constant with value true
```

```
  MYCST_NOFLAG = false        ! Constant with value false
 end-declarations

end-package
```

CHAPTER 2

User-defined subroutines

It is possible to define subroutines within a Mosel (`.mos`) program. However, in certain cases it may be preferrable to implement subroutines in the form of a module:

- An implementation of this function in C exists already.
- The subroutine manipulates data structures that are not supported by Mosel or accesses low-level (system) functions that are not available in Mosel.
- The subroutine is time-critical and must be executed as fast as possible.

2.1 Example

Some users of Mosel are annoyed by the fact that after solving an optimization problem they have to retrieve the solution value for every variable separately using function `getsol`. We therefore show in this example how to write a module `solarray` providing a procedure that copies the solution values of an array of variables into an array of reals. The arrays may be static or dynamic and of any number of dimensions (but of course, the solution array must correspond to the array of variables). Our aim is to be able to write a model along the following lines (assuming that the new procedure is also called `solarray`):

```
model "test solarray module"
 uses "solarray", "mmxprs"

 declarations
  R1=1..2
  R2={6,7,9}
  x: array(R1,R2) of mpvar
  sol: array(R1,R2) of real
 end-declarations
 ...
 solarray(x,sol)
 writeln(sol)

end-model
```

2.2 Structures for passing information

Our module needs to do the following:

- retrieve any necessary information from Mosel
- initialize itself

- define the new subroutine
- pass the new subroutine on to Mosel

To start with, we shall look at the structures that are required for exchanging information.

2.2.1 List of subroutines

The *library function* that implements the new subroutine will be called `ar_getsol`. This function and a standardized description of the subroutine it implements must be put into a *list of subroutines* that is passed to Mosel:

```
static XPRMdsofct tabfct[]=
    {
     {"solarray", 1000, XPRM_TYP_NOT, 2, "A.vA.r", ar_getsol}
    };
```

The entries of the subroutine description are the following:

- the name of the new subroutine (in a Mosel program),
- its order number within the module (not less than 1000),
- the type of the return value (here: none, we implement a procedure),
- the number and type(s) of the parameters (here: `A.v`: an array of variables and `A.r`: an array of reals), and
- the name of the C function that implements it.

A complete description of the possible values for the entries of this list is given in Section A.2.2.

2.2.2 Interface structure

The list of subroutines in turn needs to be put into the *interface structure*. Since no constants, services or types are defined by this module all other entries of this structure remain empty:

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

2.2.3 Initialization function

The module *initialization function* is almost the same as in the previous example, except for its name which must correspond to the name of the module:

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

Note that in this example — as opposed to the previous one — we are going to use functions of the Native Interface and therefore need to obtain the list of these functions from Mosel (`mm` is of type `XPRMnifct`).

2.3 Implementing the new subroutine

We now implement the new subroutine, which has to perform the following steps:

- Get the variable and solution arrays from the stack.
- Check whether the arrays are correct: verify the types, compare the array sizes and the indexing sets.
- Get the solution for all variables and copy it into the solution array.

The prototype of any library function that implements a subroutine or operator (that is, anything that is passed to Mosel via the list of subroutines structure `XPRMdsofct`) is fixed by Mosel:

```
int functionname(XPRMcontext ctx, void *libctx);
```

The first argument is the context of Mosel, the second the context of the module (see Section 2.4.1 for further detail). This module does not define its own context, we therefore do not use this parameter. The return value of the function indicates whether it was executed successfully.

The prescribed prototype of the library function does not allow any parameters to be passed directly; instead, these must be obtained from the *stack* of Mosel (see Section 2.4.2 for details). In the present case, the stack is accessed via the macro `XPRM_POP_REF`, meaning that a reference (here: array pointer) is taken from the stack. The parameter values always must be taken in the same order as they appear in the subroutine in the Mosel program.
When the library function implements a function, its return value must be put onto the stack. Since in our example we want to implement a procedure, there is no return value.

Here is the code of the module. For clarity's sake we omit the error handling in function `ar_getsol`. The same example complete with error handling, is provided with the module examples of the Mosel distribution.

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

#define MAXDIM 20

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

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

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

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

/* Module initialization function */
```

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

 return 0;
}

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

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

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

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

 return XPRM_RT_OK;
}
```

2.4 Contexts and the Mosel stack

The implementation of a new subroutine (function `ar_getsol` in the previous section) introduces several notions that may require further explanation: the Mosel and module contexts and the Mosel stack.

2.4.1 Mosel and module contexts

Any library function that implements a subroutine (or operator, as shown later in this document) takes as arguments the Mosel and the module contexts. The *Mosel context* communicates the current state of the Mosel program in question. This is necessary because several models may be executed simultaneously. Consequently, most functions of the Native Interface take the Mosel context as their first argument.

A *module* may also have a context of its own. The context of a module may be any structure that saves information about the current state of the module. Defining a module context becomes necessary when any information needs to be preserved between different calls to functions of the module during the execution of a model. In the examples discussed so far in this document (definition of constants and subroutines) this is not the case, so we do not use this parameter. Typical uses for a module context are to save the current values of control parameters published by the module or to keep track of memory allocated by the module during the execution of a model so that it may be freed at its termination. In the following chapters we give examples of these uses.

2.4.2 Working with the Mosel stack

In the case of a C library function that defines a subroutine for the Mosel language, we need to obtain the values of its parameters that have been specified in the model. The prototype for such library functions as fixed by Mosel does not allow any parameters to be passed directly; instead, the parameter values, and also the return value (if the implemented subroutine is a function), are communicated via the stack of Mosel.

The stack is accessed via the *stack access macros* `XPRM_POP_`*type* where *type* is one of

`INT` an integer or Boolean (C type int),

`REAL` a real value (C type double),

`STRING` a string (C type const char*),

`REF` any reference.

The parameter values need to be taken in the same order as they appear in the subroutine in the Mosel program. For example, if we want to implement a procedure `do_something` with the following prototype

```
procedure do_something(val1:real, num:integer, arr:array(range) of mpvar,
                       val2:real)
```

we need to take the parameters in the following order from the stack (`ctx` is the Mosel context):

```
 XPRMarray arr;
 int i;
 double r1,r2;

 r1=XPRM_POP_REAL(ctx);
 i=XPRM_POP_INT(ctx);
 arr=XPRM_POP_REF(ctx);
 r2=XPRM_POP_REAL(ctx);
```

In the example above where we implement a procedure, there is no return value. In the case of a function, the returned value must be put onto the stack using another type of stack access macro: `XPRM_PUSH_`*type* where *type* is one of the 4 types listed above. To implement a function with the prototype

```
function return_two:integer
```

that simply returns the integer value 2, we write the following:

```
static int my_return_two(XPRMcontext ctx,void *libctx)
{
 XPRM_PUSH_INT(ctx, 2);
 return XPRM_RT_OK;
}
```

2.5 Module vs. package

An implementation of the `solarray` procedure by a package is given in Chapter 'Packages' of the *Mosel User Guide*. An advantage of this package version clearly is a less technical implementation that focusses on the required functionality without any programming overhead such as the various data structures used for communication or the module initialization function. However, whilst at the C level we simply check that the two arguments have the same index sets without

having to include any more precise information about the nature of these indices, within the Mosel language the type and number of the array index sets must be known. As a consequence we have to provide a separate implementation for every case that we wish to use (one-, two-, three-,...,n-dimensional arrays indexed by integers, strings,...), restricting the functionality defined by the package to those versions that are explicitly defined.

CHAPTER 3

Creating external types

Mosel modules may create new types (referred to as *external* as opposed to the default types that are internal to Mosel), for instance other types of variables to be handled by specific solution algorithms, structures regrouping data items, or additional types of numbers. To be able to work with a new type in a Mosel program, it is not sufficient simply to define this type in a module. The module must also define all actions that one wants to be able to apply to objects of this type: creation, initialization, assignment, deletion, arithmetic operations and comparisons are typical examples. Once a new type has been created, it is treated just like a genuine type of Mosel, *e.g.* it becomes possible to define arrays and sets of this type or to use it as a function parameter.

3.1 Example

In its present version, Mosel does not allow the user to define data structures with entries of different types. In certain cases it may nevertheless be useful to organize data in such a way. Taking the example of scheduling problems, a typical group of inhomogeneous data are those related to a *task*. In our example, we shall define a structure `task` that holds the following pieces of information:

- task name (a string)
- duration (real value)
- a special flag (Boolean)
- due date (integer value)

The following model may give an overview on the types of operations and specific access functions that we have to define in order to be able to work satisfactorily with this new type:

```
model "test task module"
 uses "task"

 declarations
  R:set of integer
  t:array(R) of task
  s:task
 end-declarations

! Assigning a task
 s:=task("zero",1.5,true,3)

! Initializing a task array from file
 initializations from "testtask.dat"
  t
 end-initializations
```

```
! Reassigning the same task
 t(1):=task("one",1,true,3)
 t(1):=task("two",1,true,3)

! Various ways of creating tasks
 t(3):=task("three",10)
 t(7):=task(7)
 t(6):=task("six")
 t(9):=task(3,false,9)

! Writing a task array to file
 initializations to "testtask.dat"
  t as 't2'
 end-initializations

! Printout
 writeln("s:", s)
 writeln("t:", t)

! Accessing (and changing) detailed task information
 forall(i in R)
  writeln(i," Task ",strfmt(t(i).name,-5),": duration:", t(i).duration,
          ", flag:", t(i).aflag, ", due date:", t(i).duedate )
 t(7).name:="seven"
 t(6).duration:=4.3
 t(9).aflag:=true
 t(7).duedate:=10

! Comparing tasks
 if t(1)<>s then
  writeln("Tasks are different.")
 end-if
 t(0):=task("zero",1,true,3)
 if t(0)=s then
  writeln("Tasks are the same.")
 end-if

end-model
```

3.2 Structures for passing information

The module that we are about to write needs to provide the following:

- definition of the new type
- functions and operations on this new type, namely
  - creation and initialization functions for the new type
  - a set of subroutines for accessing (and changing) detailed task information
  - functions for reading and printing or outputting to file
  - comparison operation between tasks
- a reset service
- initialization of the module

We shall first look at the structures that must be defined for passing to Mosel the information provided by the module.

3.2.1 List of types

A *type definition* in Mosel has the following form:

```
static XPRMdsotyp tabtyp[]=
   {
    {"task", 1, XPRM_DTYP_PNCTX|XPRM_DTYP_RFCNT,
     task_create, task_delete, task_tostr, task_fromstr, task_copy, task_compare}
   };
```

The arguments given in the definition of the new type are

- the name of the new type,
- a reference number to this type within the module followed by another integer encoding type properties (here: enable calls to `task_tostr` with NULL context and indicate that the type implements reference counting);
- the six type-related functions: the first, the type instance creation function, is required whereas the remaining five: deletion, converting to string, initializing from string, copying and comparison, are optional.

A complete description of the possible values for the entries of this structure is given in Section A.2.3.

3.2.2 List of subroutines

To be able to work with this new type as shown in the model example in the previous section we have to define a *list of subroutines* as follows:

```
static XPRMdsofct tabfct[]=
    {
     {"getname", 1000, XPRM_TYP_STRING, 1, "|task|", task_getname},
     {"getduration", 1002, XPRM_TYP_REAL, 1, "|task|", task_getdur},
     {"getaflag", 1003, XPRM_TYP_BOOL, 1, "|task|", task_getaflag},
     {"getduedate", 1004, XPRM_TYP_INT, 1, "|task|", task_getdue},
     {"setname", 1005, XPRM_TYP_NOT, 2, "|task|s", task_setname},
     {"setduration", 1006, XPRM_TYP_NOT, 2, "|task|r", task_setdur},
     {"setaflag", 1007, XPRM_TYP_NOT, 2, "|task|b", task_setaflag},
     {"setduedate", 1008, XPRM_TYP_NOT, 2, "|task|i", task_setdue},
     {"@&", 1011, XPRM_TYP_EXTN, 1, "task:|task|", task_clone},
     {"@&", 1012, XPRM_TYP_EXTN, 1, "task:s", task_new1},
     {"@&", 1013, XPRM_TYP_EXTN, 1, "task:r", task_new2},
     {"@&", 1014, XPRM_TYP_EXTN, 2, "task:sr", task_new3},
     {"@&", 1015, XPRM_TYP_EXTN, 4, "task:srbi", task_new4},
     {"@&", 1016, XPRM_TYP_EXTN, 3, "task:rbi", task_new5},
     {"@:", 1020, XPRM_TYP_NOT, 2, "|task||task|", task_assign},
     {"@=", 1021, XPRM_TYP_BOOL, 2, "|task||task|", task_eql}
    };
```

Some of the notations used in this list are new and may require an explanation. The first eight subroutine definitions (get... and set...) are similar to the subroutine definition we have seen in the previous chapter:

```
{"getname", 1000, XPRM_TYP_STRING, 1, "|task|", task_getname},
```

defines the function `getname` that returns a string and takes a single argument, namely a task. The line

```
{"setname", 1005, XPRM_TYP_NOT, 2, "|task|s", task_setname},
```

defines a procedure (no return value!) that takes two arguments, a task (`|task|`) and a string (`s`). The names of external types must be surrounded by '|' in the parameter format encoding to distinguish them clearly from the one-letter encoding of Mosel's own types.

The remaining entries in the list of subroutines have special names starting with the symbol '@': they define *operators*:

`@&` constructors

`@:` assignment operator

`@=` comparison operator

The *constructors* return new objects of an external type (return code `XPRM_TYP_EXTN`). Since a module could specify several new types, the exact return type must be indicated in the format string, separated by a colon from the list of argument types.
The *assignment operator* ':' has a predefined format, as does the *comparison operator* '='.

As may be deduced from the list above, the reference numbers of the functions within the module must be in ascending order, but need not necessarily be consecutive numbers.

3.2.3 List of services

In this example, for the first time, we need to define a *service*. A service function is called by Mosel at certain predefined places (it has no direct correspondence in Mosel programs). The service function that needs to be defined when working with new types is a *reset* function. It is also required in any other cases where between several calls to module functions something needs to be kept in memory (the *context* of the module). The reset service is called at the beginning and the termination of the execution of a Mosel program that uses the module. At its first call, the reset function creates and initializes a context for the model, and deletes this context (and any other resources used by the module for this model) at the second call.

```
static XPRMdsoserv tabserv[]=
   {
    {XPRM_SRV_RESET, (void *)task_reset}
   };
```

The entry in the list of services simply indicates the type of service that is provided (here: reset) and the name of the library function that implements it.

3.2.4 Interface structure

The *interface structure* of this example defines all but the first entry with the lists of functions, types, and services shown above.

```
static XPRMdsointer dsointer=
    {
     0, NULL,
     sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
     sizeof(tabtyp)/sizeof(XPRMdsotyp), tabtyp,
     sizeof(tabserv)/sizeof(XPRMdsoserv), tabserv
    };
```

3.2.5 Module context

As mentioned earlier, the *task* module defines a *context* to collect all objects that have been created by this module during the execution of a model so that all allocated space may be freed

when the execution is terminated. In this example, the context is nothing but a chained list of tasks:

```
typedef struct
    {
     s_task *firsttask;
    } s_taskctx;
```

A module context can also be used to store the current values of control parameters (see Chapter 4) or any other information that needs to be preserved between different calls to the module functions during the execution of a model.

3.3 Type-related functions

In this example, the following structure represents a *task*:

```
typedef struct Task
     {
      int refcnt;
      const char *name;
      int aflag, duedate;
      double duration;
      struct Task *next;
     } s_task;
```

The first entry of this structure is the *reference counter* (with the flag `XPRM_DTYP_RFCNT` set at the type definition we have indicated that our module implements reference counting for the type 'task'). The next four entries of this structure correspond directly to the information associated with a task (name, a Boolean flag, due date, duration). The last entry (`next`) points to the following element in the list of tasks held by the module context.

In the definition of the new type `task`, we have indicated the names of 5 functions for creating and deleting the new type, getting a textual representation and initializing the new type from a textual representation, and copying the type. The only function that is always required for any type definition is the creation function, the remaining ones are optional (for the deletion function depending on the type properties).

3.3.1 Type creation and deletion

The objective of the type instance creation and deletion functions is to handle (create/initialize or delete/reset) the C structures that represent the external type and to update correspondingly the information stored in the module context. In this example we implement just a rudimentary memory management for the objects (tasks) created by the module: every time a task is created, we allocate the corresponding space and deallocate it when the task is deleted. In Chapter 5 a more realistic example is given that allocates chunks of memory and recycles space that has been allocated earlier by the module.

Reference counting: the flag `XPRM_DTYP_RFCNT` set at the type definition indicates that our module handles reference counting for the type `task`. As a consequence Mosel may call the type creation function with a reference to a previously created object for increasing its reference count. The type deletion function (which is mandatory in this case) is called as many times as the creation function has been used for a given object before this object is effectively released.

We define the task *creation function* as follows:

```
static void *task_create(XPRMcontext ctx, void *libctx, void *todup,
                         int typnum)
```

```
{
 s_taskctx *taskctx;
 s_task *task;

 if(todup!=NULL)
 {
  ((s_task *)todup)->refcnt++;
  return todup;
 }
 else
 {
  taskctx=libctx;
  task=(s_task *)malloc(sizeof(s_task));
  task->next=taskctx->firsttask;
  taskctx->firsttask=task;
  task->refcnt=1;

  task->name=NULL;                    /* Initialize the task */
  task->duration=0;
  task->aflag=task->duedate=0;
  return task;
 }
}
```

The task *deletion function* frees the space used by a task and removes the task from the list of tasks held by the module context if no reference to the task is left. Otherwise, it decreases the reference counter. If the task is not found in the list we display an error message using the Native Interface function `dispmsg`. For any output produced by modules, this way of printing should always be preferred to the corresponding C printing functions.

```
static void task_delete(XPRMcontext ctx, void *libctx, void *todel,
                        int typnum)
{
 s_taskctx *taskctx;
 s_task *task,*prev;

 if((todel!=NULL)&&((--((s_task *)todel)->refcnt)<1))
 {
  taskctx=libctx;
  task=todel;
  if(taskctx->firsttask==task) taskctx->firsttask=task->next;
  else
  {
   prev=taskctx->firsttask;
   while((prev->next!=NULL) && (prev->next!=task))
    prev=prev->next;
   if(prev->next==NULL) mm->dispmsg(ctx, "Task: task not found.\n");
   else prev->next=task->next;
  }
  free(task);
 }
}
```

The definition of a type instance deletion function does *not* replace the memory deallocation in the reset service function (see Section 3.4).

3.3.2 Conversion to and from string

To be able to use `initializations` blocks with the new type `task` we define two functions for transforming the task into a string and initializing it from a string. The writing function is also used by the `write` and `writeln` procedures for printing this type. The reading function also gets applied by default when the type instance creation function is given a string, but in this example we have defined that the string is interpreted only as the task name.
The format of the string will obviously depend on the type. In this example we have chosen a

very simple string format for tasks: the data entries separated by blanks in the order name, duration, flag, due date. The following function prints a task:

```
static int task_tostr(XPRMcontext ctx, void *libctx, void *toprt, char *str,
                      int len, int typnum)
{
 s_task *task;

 if(toprt==NULL)
  return 0;
 else
 {
  task=toprt;
  return snprintf(str, len, "%s %g %d %d", task->name, task->duration,
                  task->aflag, task->duedate);
 }
}
```

The next function reads in a task from a string (the flag and due date values may have been omitted):

```
static int task_fromstr(XPRMcontext ctx, void *libctx, void *toinit,
                        const char *str, int typnum, const char **endp)
{
 double dur;
 int af,due,res,cnt;
 char *name;
 s_taskctx *taskctx;
 s_task *task;

 taskctx=libctx;
 name=alloca(TASK_MAXNAME*sizeof(char));
 af=due=cnt=0;
 res=sscanf(str,"%s %lf %d%n %d%n",name,&dur,&af,&cnt,&due,&cnt);
 if(res<3)
 {
  if(endp!=NULL) *endp=str;
  return XPRM_RT_ERROR;
 }
 else
 {
  task=toinit;
  task->name=mm->regstring(ctx, name);
  task->duration=dur;
  task->aflag=(res>=3)?af:0;
  task->duedate=(res==4)?due:0;
  if(endp!=NULL) *endp=str+cnt;
  return XPRM_RT_OK;
 }
}
```

The Native Interface function `regstring` that is used here adds the name string to the names dictionary. Any string that is returned to Mosel must be registered this way.

3.3.3 The copy function

Certain assignments in Mosel (assignments that are not stated explicitly, such as array initialization) use the type copy function. If no copy function is defined for a type, the operations where it is necessary are disabled by the compiler for the corresponding type.

For copying the type `task` we may define the following function where the task `toinit` becomes a copy of the task `src`:

```
static int task_copy(XPRMcontext ctx, void *libctx, void *toinit, void *src,
```

```
                    int typnum)
{
 s_task *task1,*task2;

 task1=(s_task *)toinit;
 if(src==NULL)
 {
  task1->name=NULL;
  task1->duration=0;
  task1->aflag=task1->duedate=0;
 }
 else
 {
  task2=(s_task *)src;
  task1->name=task2->name;
  task1->aflag=task2->aflag;
  task1->duedate=task2->duedate;
  task1->duration=task2->duration;
 }
 return 0;
}
```

3.3.4 The compare function

The compare function is required for the comparison of aggregate objects (for example, a record that contains a field of type 'task'). If no compare function is defined for a type, the operations where it is necessary are disabled by the compiler for the corresponding type.

The following function compares two objects `t1` and `t2` of type `task` by comparing all the fields of the two structures. For the comparison of the names it suffices to compare the pointers because we are using the names dictionary of Mosel: it guarantees the uniqueness of the name strings.

```
static int task_compare(XPRMcontext ctx, void *libctx, void *t1, void *t2, int typnum)
{
 int b;

 if(t1!=NULL)
 {
  if(t2!=NULL)
   b=((((s_task *)t1)->name==((s_task *)t2)->name)  /* This is correct since we
                                                       are using Mosel's dictionary */
   &&(((s_task *)t1)->duration==((s_task *)t2)->duration)
   &&(((s_task *)t1)->aflag==((s_task *)t2)->aflag)
   &&(((s_task *)t1)->duedate==((s_task *)t2)->duedate));
  else
   b=0;
 }
 else
  b=(t2==NULL);

 switch(XPRM_COMPARE(typnum))
 {
  case MM_COMPARE_EQ:
    return b;
  case MM_COMPARE_NEQ:
    return !b;
  default:
    return XPRM_COMPARE_ERROR;
 }
}
```

3.4 Service function `reset`

Just like the other library functions, the reset service function takes a predefined format. Here we

create the module context at the first call to this function and delete it at the subsequent call. When deleting the context the reset function needs to free all space that has been allocated by the module during the execution of a model. Therefore, every time a task is created it is added to the list of tasks in the module context and it is removed from the list if it is deleted explicitly by a call to the type instance deletion function. As mentioned earlier, even if a module provides deletion functions for all the types that it defines (as in this example) it is required to implement the reset service to free any remaining allocated space because Mosel does not guarantee that the type instance deletion function gets called for every object that has been created by the module.

```
static void *task_reset(XPRMcontext ctx, void *libctx, int version)
{
 s_taskctx *taskctx;
 s_task *task;

 if(libctx==NULL)          /* At start: create the context */
 {
  taskctx=malloc(sizeof(s_taskctx));
  memset(taskctx, 0, sizeof(s_taskctx));
  return taskctx;
 }
 else                      /* At the end: delete everything */
 {
  taskctx=libctx;
  while(taskctx->firsttask!=NULL)
  {
   task=taskctx->firsttask;
   taskctx->firsttask=task->next;
   free(task);
  }
  free(taskctx);
  return NULL;
 }
}
```

3.5 Other library functions and operators

The list of subroutines contains several groups of subroutines that may be applied to the new type `task`:

- constructor functions (cloning and initialization with data)
- subroutines for accessing detailed task information (getting and setting name, duration *etc.*)
- assignment and comparison of tasks

3.5.1 Constructors

Being able to *clone a type* is required in certain cases of assignments (the use is similar to the cloning operation in C++):

```
static int task_clone(XPRMcontext ctx, void *libctx)
{
 s_task *task, *new_task;

 task=XPRM_POP_REF(ctx);
 if(task!=NULL)
 {
  new_task=task_create(ctx, libctx, NULL, 0);
  new_task->name=task->name;
  new_task->aflag=task->aflag;
```

```
  new_task->duedate=task->duedate;
  new_task->duration=task->duration;
  XPRM_PUSH_REF(ctx, new_task);
 }
 else
  XPRM_PUSH_REF(ctx, NULL);
 return XPRM_RT_OK;
}
```

As may be deduced from the test performed in this function, Mosel may pass the NULL pointer to a function in the place of an external type. This will typically happen if the object is an entry of a dynamic array that has not been initialized.

The following is an example of a *constructor function*. It creates a new task and fills it with the given data. This function enables the user to create a task by writing for example:

```
task("a_task", 3.5, true, 10)
```

Several overloaded versions of this function are defined in our example. They are similar to this one and we omit printing them here. In every case, all given information needs to be taken from the stack and the reference to the new task is put back onto the stack.

```
static int task_new4(XPRMcontext ctx, void *libctx)
{
 s_task *task;

 task=task_create(ctx, libctx, NULL, 0);
 task->name=XPRM_POP_STRING(ctx);
 task->duration=XPRM_POP_REAL(ctx);
 task->aflag=XPRM_POP_INT(ctx);
 task->duedate=XPRM_POP_INT(ctx);
 XPRM_PUSH_REF(ctx, task);
 return XPRM_RT_OK;
}
```

3.5.2 Accessing detailed task information

We only give one example of a function for retrieving detailed task information (namely the task name), the other three are very similar:

```
static int task_getname(XPRMcontext ctx, void *libctx)
{
 s_task *task;

 task=XPRM_POP_REF(ctx);
 if(task==NULL)
 {
  mm->dispmsg(ctx, "Task: Accessing undefined task.\n");
  return XPRM_RT_ERROR;
 }
 XPRM_PUSH_STRING(ctx, task->name);
 return XPRM_RT_OK;
}
```

The following is an example of a function that sets some detailed task information (namely the duration):

```
static int task_setdur(XPRMcontext ctx, void *libctx)
{
 s_task *task;
 double dur;

 task=XPRM_POP_REF(ctx);
```

```
 dur=XPRM_POP_REAL(ctx);
 if(task==NULL)
 {
  mm->dispmsg(ctx, "Task: Accessing undefined task.\n");
  return XPRM_RT_ERROR;
 }
 task->duration=dur;
 return XPRM_RT_OK;
}
```

Since the names of the task access functions defined by our module adhere to the standard Mosel naming scheme (`get`*property* and `set`*property*) Mosel deduces automatically the dot notation for tasks. That means that for a task `t` we may use equivalently, for instance, `getname(t)` and `t.name` or `setduration(t,10)` and `t.duration:=10`.

3.5.3 Assignment and comparison operators

The *assignment operation* takes two task references from the stack, assigns the second to the first and deletes the second task since this is only an intermediate object:

```
static int task_assign(XPRMcontext ctx, void *libctx)
{
 s_task *task1, *task2;

 task1=XPRM_POP_REF(ctx);
 task2=XPRM_POP_REF(ctx);
 task1->name=task2->name;
 task1->aflag=task2->aflag;
 task1->duedate=task2->duedate;
 task1->duration=task2->duration;
 task_delete(ctx, libctx, task2, 0);
 return XPRM_RT_OK;
}
```

The implementation of the *comparison* operation for two tasks compares all the fields of the two structures, similarly to the implementation of the type comparison function that we have seen above in Section 3.3.

```
static int task_eql(XPRMcontext ctx, void *libctx)
{
 s_task *task1, *task2;
 int b;

 task1=XPRM_POP_REF(ctx);
 task2=XPRM_POP_REF(ctx);
 if(task1!=NULL)
 {
  if(task2!=NULL)
   b=((task1->name==task2->name) && (task1->duration==task2->duration)
     && (task1->aflag==task2->aflag) && (task1->duedate==task2->duedate));
  else
   b=0;
 }
 else
  b=(task2==NULL);
 XPRM_PUSH_INT(ctx,b);
 return XPRM_RT_OK;
}
```

Note that once we have defined the equality comparison, there is no need to implement the difference-between-tasks operation: it is derived by Mosel as being the negation of the equality.

3.6 Module vs. package

With Mosel Release 2 it has become possible to define new user types directly in the Mosel language. An equivalent definition of the type 'task' within a package is the following.

```
public declarations
 task = public record
  name: string
  duration: real
  aflag: boolean
  duedate: integer
 end-record
end-declarations
```

The access functions `get`... and `set`... may be defined to work exactly in the same way as those defined by our module. However, if we work with the dot notation to access the record fields the definition of these functions is not required. The type `task` defined by a package will use the standard conventions of Mosel for reading and writing records from/to a file—in a module these subroutines must be defined explicitly, which also implies that they are not confined to the standard Mosel format for reading and writing records.

A package cannot provide constructors for tasks, instead it might define subroutines to initialize (existing) tasks with data, for example, replacing the line

```
t(9):=task(3,false,9)
```

in our test model from Section 3.1 by

```
create(t(9))
inittask(t(9), 3, false, 9)
```

Another feature that is not supported by packages is the definition of operators. The (default) comparison of two tasks defined through a package such as `t(1)<>s` compares whether we are looking at the same object (*i.e.*, same address in memory)—the field-wise comparison of the contents of tasks needs to be implemented differently, for instance, by a subroutine `issame(t(1), s)`.

To summarize the above, it is possible to implement all the functionality of the *task* module by the means of a package, requiring less programming effort where we rely on standard Mosel features (in particular for reading/writing types) at the expense of some flexibility. However, since same functionality does not mean same way of functioning the choice of the package or the module version of the type definition makes necessary certain modifications to the Mosel model that uses the respective library.

CHAPTER 4

Control parameters

Control parameters may be used to direct and modify the behaviour of modules or to obtain status information from a module. A module may provide such parameters as read-only, for information purposes. But much more frequently the control parameters will be write-enabled, giving the user the possibility to modify their value.

4.1 Example

We want to add two parameters to the module defining a task structure that was presented in the previous chapter: the maximum length of name strings used for reading in tasks (`tasknamelength`, an integer value) and a time limit value (`taskmaxtime`, a real). These parameters might be used as follows in a model (assuming `t` is an array of tasks):

```
if(getparam("tasknamelength")<10) then
 setparam("tasknamelength",20)
end-if

t(3):=task("three",getparam("taskmaxtime"))
```

4.2 Structures for passing information

The introduction of parameters necessitates several additions to the lists that are passed to Mosel via the interface structure.

4.2.1 List of subroutines

In the *list of subroutines*, the following two lines are new (they must be added at the beginning of the list and in the order shown here):

```
static XPRMdsofct tabfct[]=
   {
    {"", XPRM_FCT_GETPAR, XPRM_TYP_NOT, 0, NULL, task_getpar},
    {"", XPRM_FCT_SETPAR, XPRM_TYP_NOT, 0, NULL, task_setpar},
    ...
   }
```

These two subroutines do not take any names (first parameter). The macros `XPRM_FCT_GETPAR` and `XPRM_FCT_SETPAR` identify them as implementations of Mosel's `getparam` and `setparam` subroutines for this module.

4.2.2 List of services

We have also got two new *services*:

```
static XPRMdsoserv tabserv[]=
   {
    {XPRM_SRV_RESET, (void *)task_reset},
    {XPRM_SRV_PARAM, (void *)task_findparam},
    {XPRM_SRV_PARLST, (void *)task_nextparam}
   };
```

4.2.3 Module context

The user is free to store the *control parameters* in any way that is convenient for him. There is no predefined format for this list since it is not passed as such to Mosel. In our example we have chosen the following structure for storing parameters (their names — always in lower case only, types and access rights, and descriptions):

```
static struct
{
 char *name;
 int type;
 char *desc;
} taskparams[]={
 {"taskmaxtime", XPRM_TYP_REAL|XPRM_CPAR_READ|XPRM_CPAR_WRITE,
  "a time limit value"},
 {"tasknamelength", XPRM_TYP_INT|XPRM_CPAR_READ|XPRM_CPAR_WRITE,
  "maximum length of task names"}
};
```

The current values of the parameters are stored in the context of the module since they may be modified (these values must be initialized when the context is created):

```
typedef struct
    {
     s_task *firsttask;
     int maxname;
     double maxtime;
    } s_taskctx;
```

4.3 Services related to parameters

Whenever a module defines control parameters, it needs to provide the service to retrieve a parameter number by a name. If the corresponding parameter is not found in the module, this function returns -1. Otherwise, if the parameter belongs to the module, its reference number (here: index in the list of parameters defined by the module) must be returned, together with information about its type (second argument of the function).

```
static int task_findparam(const char *name, int *type)
{
 int n;
 int notfound;

 n=0;
 do
 {
  if((notfound=strcmp(name, taskparams[n].name))==0) break;
  n++;
 } while(taskparams[n].name!=NULL);
```

```
 if(!notfound)
 {
  *type=taskparams[n].type;
  return n;
 }
 else
  return -1;
}
```

The `findparam` service function is only used during the compilation of a model to convert the name of a parameter to a module-internal identification number. This number is used by the subroutines `setparam` and `getparam` during the execution of the model (see Section 4.4).

The second service that we are defining is optional: it provides a possibility of enumerating the parameters of the module (*e.g.* this is used when module information is displayed with the `examine` command).

```
static void *task_nextparam(void *ref, const char **name, const char **desc,
                            int *type)
{
 long cst;

 cst=(long)ref;
 if((cst<0)||(cst>=TASK_NUMPARAM))
  return NULL;
 else
 {
  *name=taskparams[cst].name;
  *type=taskparams[cst].type;
  *desc=taskparams[cst].desc;
  return (void *)(cst+1);
 }
}
```

Mosel calls this function repeatedly until it returns `NULL`. At the first call the value of the argument `ref` is `NULL`, while at any subsequent calls it corresponds to the return value of the immediately preceding execution of this function. The other arguments need to be filled with the information for a parameter (name and type are required, the descriptive text is optional). The constant `TASK_NUMPARAM` is the number of parameters that we have defined in this module.

4.4 Functions for handling parameters

In a Mosel program, parameters are accessed with the two subroutines `setparam` and `getparam`. The module must implement these two subroutines for its parameters.
The function that enables the user to set the parameters of our module is the following:

```
static int task_setpar(XPRMcontext ctx, void *libctx)
{
 s_taskctx *taskctx;
 int n;

 taskctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0: taskctx->maxname=XPRM_POP_INT(ctx); break;
  case 1: taskctx->maxtime=XPRM_POP_REAL(ctx); break;
  default: mm->dispmsg(ctx, "Task: Wrong control parameter number.\n");
   return XPRM_RT_ERROR;
 }
 return XPRM_RT_OK;
}
```

Via its stack, Mosel provides the number of the parameter (value returned by the `findparam` service function) and its new value to the module.

The parameters of our module are accessed via the following function:

```
static int task_getpar(XPRMcontext ctx, void *libctx)
{
 s_taskctx *taskctx;
 int n;

 taskctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0: XPRM_PUSH_INT(ctx, taskctx->maxname); break;
  case 1: XPRM_PUSH_REAL(ctx, taskctx->maxtime); break;
  default: mm->dispmsg(ctx, "Task: Wrong control parameter number.\n");
   return XPRM_RT_ERROR;
 }
 return XPRM_RT_OK;
}
```

The complete *task* module is part of the module examples provided with the Mosel distribution and on the Xpress website.

4.5 Module vs. package

Control parameters can only be implemented by modules, packages do not offer any corresponding functionality.

CHAPTER 5

Creating external types: second example

Mosel defines the types `integer`, `real` and `boolean` on which arithmetic operations may be used. By creating modules it is possible to add other types, such as complex numbers, to this list. In the previous chapters we have already seen an example of how to define a new type in a module, but this new type `task` was not suited to be used with arithmetic operations. In this chapter we shall therefore give another example of the definition of a type, this time of a type to which such operations may sensibly be applied.

5.1 Example

In this chapter we are going to define the type `complex` to represent complex numbers. The following example demonstrates the typical uses that one may wish to make of a mathematical type like complex numbers in a model:

- use of data structures
- various types of initializations and assignments
- products, sums and other arithmetic operations
- comparison
- printed output on screen and to a file.

The following model shows how one might work with a new type `complex` in Mosel:

```
model "Test complex"
 uses "complex"

 declarations
  c:complex
  t:array(1..10) of complex
 end-declarations

 forall(j in 1..10) t(j):=complex(j,10-j)
 t(5):=complex("5+5i")

 c:=prod(i in 1..5) t(i)
 if c<>0 then
  writeln("product: ",c)
 end-if

 writeln("sum: ", sum(i in 1..10) t(i))
 c:= t(1)*t(3)/t(4) + if(t(2)=0,t(10),t(8)) + t(5) - t(9)
 writeln("result: ", c)
```

```
 initializations to "test.dat"
  c t
 end-initializations

end-model
```

5.2 Structures for passing information

Complex numbers are usually represented as *a* + *bi* where *a* and *b* are real numbers. *a* is called the real part and *bi* the imaginary part. We implement the following C structure to store a complex number:

```
typedef struct
    {
     int refcnt;              /* For reference count */
     double re, im;           /* Real and imaginary parts */
    } s_complex;
```

5.2.1 List of subroutines

The main interest of this example lies in the definition of its *list of subroutines* which actually is a list of operators:

```
static XPRMdsofct tabfct[]=
    {
     {"@&", 1000, XPRM_TYP_EXTN, 1, "complex:|complex|", cx_new0},
     {"@&", 1001, XPRM_TYP_EXTN, 1, "complex:r", cx_new1},
     {"@&", 1002, XPRM_TYP_EXTN, 2, "complex:rr", cx_new2},
     {"@0", 1003, XPRM_TYP_EXTN, 0, "complex:", cx_zero},
     {"@1", 1004, XPRM_TYP_EXTN, 0, "complex:", cx_one},
     {"@:", 1005, XPRM_TYP_NOT,  2, "|complex||complex|", cx_asgn},
     {"@:", 1006, XPRM_TYP_NOT,  2, "|complex|r", cx_asgn_r},
     {"@+", 1007, XPRM_TYP_EXTN, 2, "complex:|complex||complex|", cx_pls},
     {"@+", 1008, XPRM_TYP_EXTN, 2, "complex:|complex|r", cx_pls_r},
     {"@*", 1009, XPRM_TYP_EXTN, 2, "complex:|complex||complex|", cx_mul},
     {"@*", 1010, XPRM_TYP_EXTN, 2, "complex:|complex|r", cx_mul_r},
     {"@-", 1011, XPRM_TYP_EXTN, 1, "complex:|complex|", cx_neg},
     {"@/", 1012, XPRM_TYP_EXTN, 2, "complex:|complex||complex|", cx_div},
     {"@/", 1013, XPRM_TYP_EXTN, 2, "complex:|complex|r", cx_div_r1},
     {"@/", 1014, XPRM_TYP_EXTN, 2, "complex:r|complex|", cx_div_r2},
     {"@=", 1015, XPRM_TYP_BOOL, 2, "|complex||complex|", cx_eql},
     {"@=", 1016, XPRM_TYP_BOOL, 2, "|complex|r", cx_eql_r}
    };
```

In the order of their appearance this list defines the following operators:

`@&` creation (construction)

`@0` zero element for sums

`@1` one element for products

`@:` assignment

`@+` addition

`@*` multiplication

`@-` negation

`@/` division

`@=` comparison (test of equality)

For most operators in the list above several versions are defined, with different types or combinations of types. The only type conversion that is carried out automatically by Mosel is from integer to real (but not the other way round), and no conversions involving external types. It is therefore necessary to define all the operations between two numbers for two complex numbers and also for a complex and a real number. For commutative operations (addition, multiplication, comparison) it is only required to define one version combining the two types, the other sense is deduced by Mosel: for example, if *complex* + *real* is defined, Mosel 'knows' how to calculate *real* + *complex*. For division (not commutative) we need to define every case separately.

5.2.2 List of types

The definition of the new type in the list of types that is passed to Mosel looks as follows:

```
static XPRMdsotyp tabtyp[]=
    {
     {"complex", 1, XPRM_DTYP_PNCTX|XPRM_DTYP_RFCNT|XPRM_DTYP_APPND,
      cx_create, cx_delete, cx_tostr, cx_fromstr, cx_copy, cx_compare}
    };
```

The type-related functions (`cx_create`: creation, `cx_delete`: deletion, `cx_tostr`: transformation to a string, `cx_fromstr`: initialization from a string, `cx_copy`: copying, `cx_compare`: comparison) could be implemented in a similar way to what has been shown for the *task* module in the previous chapters. But, for practical purposes, this rudimentary memory management may not be efficient enough. In this chapter we therefore give an example of improved memory management for external types. This includes new versions of the type instance creation and deletion functions, an adaptation of the reset service, and the definition of additional list structures for storing information in the module context.

The functions for converting types to or from strings and also the copy and compare functions described for the *task* module only require minor modifications to adapt them to this example. Their definition will not be repeated in this chapter.
The list of services (merely consisting of the reset service) and the main interface structure are also very similar to those of the *task* module, and the module initialization function remains the same except for its name. We therefore refrain from printing them here.

The complete source code of the *complex* module is among the module examples provided with the Mosel distribution and on the Xpress website.

5.3 Definition of operators

In this section we show several examples of the implementation of operators. A comprehensive list of all operators that may be defined in Mosel is given in the appendix.

5.3.1 Constructors

In the chapter about the *task* module we have already seen examples of functions for cloning a new type and constructing it in different ways. Here the cloning operation is implemented as follows:

```
static int cx_new0(XPRMcontext ctx, void *libctx)
{
 s_complex *complex,*new_complex;
```

```
 complex=XPRM_POP_REF(ctx);
 if(complex!=NULL)
 {
  new_complex=cx_create(ctx, libctx, NULL, 0);
  *new_complex=*complex;
  XPRM_PUSH_REF(ctx, new_complex);
 }
 else
  XPRM_PUSH_REF(ctx, NULL);
 return XPRM_RT_OK;
}
```

A new complex number is constructed from two given real numbers thus:

```
static int cx_new2(XPRMcontext ctx, void *libctx)
{
 s_complex *complex;

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

5.3.2 Comparison operators

Another operation that we have already seen in the *task* module is the comparison between new types. This can be done in a very similar way for module `complex` and is not repeated here. In addition, it makes sense to define a comparison between a complex and a real number:

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

 c1=XPRM_POP_REF(ctx);
 r=XPRM_POP_REAL(ctx);
 if(c1!=NULL)
  b=(c1->im==0)&&(c1->re==r);
 else
  b=(r==0);
 XPRM_PUSH_INT(ctx,b);
 return XPRM_RT_OK;
}
```

5.3.3 Arithmetic operators

The arithmetic operations must implement the rules to perform these operations on complex numbers.

5.3.3.1 Multiplication

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

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

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

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

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

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

It is not necessary to define the multiplication of a real with a complex since this operation is commutative and Mosel therefore deduces this case.

5.3.3.2 Addition, subtraction, division

The *addition* of two complex numbers and of a complex and a real number is implemented in a very similar way to multiplication. Once we have got the two types of addition, we simply need to implement the negation (–complex) in order for Mosel to be able to deduce *subtraction* (real – complex and complex – complex):

```
static int cx_neg(XPRMcontext ctx, void *libctx)
{
 s_complex *c1;

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

For *division*, we need to implement all three cases since this operation is not commutative: complex/complex, complex/real and real/complex. Since these functions again are similar to the

implementations of the other arithmetic operations that have been shown, they are not printed here.

5.3.3.3 Identity elements for addition and multiplication

In the list of operators printed in the previous section, there appear two more operators: `@0` and `@1`. These two generate the identity elements for addition and multiplication respectively:

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

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

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

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

5.4 Improved memory management for external types

For the *task* module we have described a very simple way of handling memory allocations in a module directly with the corresponding C functions: whenever an object of the new type needs to be created the required space is allocated and when the object is deleted this space is freed in C.

In this section we give an example of memory management by the module: the space for new complex numbers is allocated in large chunks. The module keeps track of the available space, including space that has already been used by this module and may be recycled. This proceding requires much less memory allocation operations and only a single set of deallocations. Furthermore, at the deletion of an object the possibily expensive search for the object in the entire list held by the module context is replaced by a copy of the pointer to the list of free space.

5.4.1 Module context

Contrary to the context of the *task* module that only keeps a single list, we now define a context that holds two lists:

```
typedef struct
    {
     s_nmlist *nmlist;
     u_freelist *freelist;
    } s_cxctx;
```

The first of these lists, `nmlist`, is all the space allocated for complex numbers, stored in chunks of size `NCXL`:

```
typedef struct Nmlist
    {
     s_complex list[NCXL];
```

```
     int nextfree;
     struct Nmlist *next;
    } s_nmlist;
```

The second list indicates the free entries in the list of numbers:

```
typedef union Freelist
    {
     s_complex cx;
     union Freelist *next;
    } u_freelist;
```

5.4.2 Service function `reset`

The reset service function initializes the module context at its first call and frees all space that has been allocated by the module at the next call to it:

```
static void *cx_reset(XPRMcontext ctx, void *libctx, int version)
{
 s_cxctx *cxctx;
 s_nmlist *nmlist;

 if(libctx==NULL)              /* libctx==NULL => initialization */
 {
  cxctx=malloc(sizeof(s_cxctx));
  memset(cxctx, 0, sizeof(s_cxctx));
  return cxctx;
 }
 else                          /* Otherwise release the resources we use */
 {
  cxctx=libctx;
  while(cxctx->nmlist!=NULL)
  {
   nmlist=cxctx->nmlist;
   cxctx->nmlist=nmlist->next;
   free(nmlist);
  }
  free(cxctx);
  return NULL;
 }
}
```

5.4.3 Type creation and deletion functions

In our example we define the task *creation function* printed below. As mentioned in the previous section, the space for complex numbers is not allocated one-by-one but in larger chunks and the module also keeps track of space that may be re-used. We therefore face the following choice every time a new complex number is created:

- if possible re-use space that has been allocated earlier,
- otherwise, if no free space remains, allocate a new block of complex numbers,
- otherwise use the next free space.

In the case that the complex number passed into the creation function already exists we simply augment its reference counter.

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

```
 s_complex *complex;
 s_nmlist *nmlist;

 if(todup!=NULL)
 {
  ((s_complex *)todup)->refcnt++;
  return todup;
 }
 else
 {
  cxctx=libctx;
  if(cxctx->freelist!=NULL)         /* Re-use allocated space that was freed */
  {
   complex=&(cxctx->freelist->cx);
   cxctx->freelist=cxctx->freelist->next;
  }
  else                              /* Allocate a new block of complex numbers */
   if((cxctx->nmlist==NULL)||(cxctx->nmlist->nextfree>=NCXL))
   {
    nmlist=malloc(sizeof(s_nmlist));
    nmlist->next=cxctx->nmlist;
    cxctx->nmlist=nmlist;
    nmlist->nextfree=1;
    complex=nmlist->list;
   }
   else                             /* Use allocated and yet free space */
    complex=&(cxctx->nmlist->list[cxctx->nmlist->nextfree++]);

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

The *deletion function* does not completely deallocate the space used by a complex number. It simply moves it into the list of space that may be recycled:

```
static void cx_delete(XPRMcontext ctx, void *libctx, void *todel, int typnum)
{
 s_cxctx *cxctx;
 u_freelist *freelist;

 if((todel!=NULL)&&((--((s_complex *)todel)->refcnt)<1))
 {
  cxctx=libctx;
  freelist=todel;                    /* Delete = space to be recycled */
  freelist->next=cxctx->freelist;
  cxctx->freelist=freelist;
 }
}
```

5.5 Module vs. package

Operators can only be implemented by the means of modules, it is not possible to define operators within the Mosel language (that is, packages cannot provide any corresponding functionality).

CHAPTER 6

Implementing an LP/MIP solver interface

The Mosel NI publishes a special set of functionality that provides access to the matrix-based representation of optimization problems formulated using the Mosel types `mpvar`, `linctr`, `mpproblem`, that is, LP and MIP problems. These NI functions can be used for implementing interfaces to optimization solvers that are available in the form of a C/C++ library.

6.1 Example

This chapter explains how to implement a basic Mosel module *myxprs* for using Xpress Optimizer as the solver for optimization models stated in Mosel. The use of the new module from Mosel looks as follows for its simplest form that provides starting of the solver, solution retrieval, and access to solver parameters.

```
model "Problem solving and solution retrieval"
  uses "myxprs"                       ! Load the solver module

  declarations
    x,y: mpvar                        ! Some decision variables
    pb: mpproblem                     ! (Sub)problem
  end-declarations

  procedure printsol
    if getprobstat = MYXP_OPT then
      writeln("Solution: ", getobjval, ";", x.sol, ";", y.sol)
    else
      writeln("No solution found")
    end-if
  end-procedure

  Ctr1:= 3*x + 2*y <= 400
  Ctr2:= x + 3*y <= 310
  MyObj:= 5*x + 20*y

 ! Setting solver parameters
  setparam("myxp_verbose", true)      ! Display solver log
  setparam("myxp_maxtime", 10)        ! Set a time limit

 ! Solve the problem (includes matrix generation)
  maximize(MyObj)

 ! Retrieve a solver parameter
  writeln("Solver status: ", getparam("myxp_lpstatus"))
 ! Access solution information
  printsol

 ! Turn poblem into a MIP
  x is_integer; y is_integer

 ! Solve the modified problem
```

```
  maximize(MyObj)
  printsol

 ! **** Define and solve a (sub)problem ****
  with pb do
    3*x + 2*y <= 350
    x + 3*y <= 250
    maximize(5*x + 20*y)
    printsol
  end-do

end-model
```

Sections 6.4.2 and 6.5 show how to extend this initial version with a solution callback and a matrix export subroutine including names generation.

6.2 Structures for passing information

A minimal implementation of a solver module needs to do the following:

- Modeling functionality:
  - define subroutines to start an optimization run and retrieve solution values
  - provide access to solver parameters
  - if supported by the solver, provide support for handling multiple problems
- NI functionality:
  - implement a reset and an unload service
  - initialize the module and the required interface structures

To start with, let us take a look at the structures that are required for exchanging information between Mosel and the external program.

6.2.1 List of subroutines

The minimal set of entries for the list of subroutines would be just the calls to minimization/maximization. Our implementation adds a function to retrieve the problem status information, alternative spelling for the optimization routines, and it also provides the access routines for module control parameters that are required for the implementation of solver parameters.

```
static XPRMdsofct tabfct[]=
    {
     {"",XPRM_FCT_GETPAR,XPRM_TYP_NOT,0,NULL,slvlc_getpar},
     {"",XPRM_FCT_SETPAR,XPRM_TYP_NOT,0,NULL,slvlc_setpar},
     {"getprobstat",2000,XPRM_TYP_INT,0,NULL,slvlc_getpstat},
     {"minimise",2100,XPRM_TYP_NOT,1,"c",slvlc_minim},
     {"minimize",2100,XPRM_TYP_NOT,1,"c",slvlc_minim},
     {"maximise",2101,XPRM_TYP_NOT,1,"c",slvlc_maxim},
     {"maximize",2101,XPRM_TYP_NOT,1,"c",slvlc_maxim}
    };
```

6.2.2 List of parameters

In terms of an example, we provide access to a few controls of Xpress Optimizer, and the module also shows how to implement a verbosity flag, resulting in the following list of module parameters:

```
static struct                   /* Parameters published by this module */
    {
     char *name;
     int type;
    } myxprsparams[]=
    {
     {"myxp_verbose",XPRM_TYP_BOOL|XPRM_CPAR_READ|XPRM_CPAR_WRITE},
     {"myxp_maxtime",XPRM_TYP_INT|XPRM_CPAR_READ|XPRM_CPAR_WRITE},
     {"myxp_lpstatus",XPRM_TYP_INT|XPRM_CPAR_READ},
     {"myxp_lpobjval",XPRM_TYP_REAL|XPRM_CPAR_READ},
    };
```

The problem and LP status parameters return values that are best implemented via module constants, such as:

```
static XPRMdsoconst tabconst[]=
    {
     XPRM_CST_INT("MYXP_INF",XPRM_PBINF),             /* Mosel status codes */
     XPRM_CST_INT("MYXP_OPT",XPRM_PBOPT),
     XPRM_CST_INT("MYXP_OTH",XPRM_PBOTH),
     XPRM_CST_INT("MYXP_UNF",XPRM_PBUNF),
     XPRM_CST_INT("MYXP_UNB",XPRM_PBUNB),
     XPRM_CST_INT("MYXP_LP_OPTIMAL",XPRS_LP_OPTIMAL), /* Solver status codes */
     XPRM_CST_INT("MYXP_LP_INFEAS",XPRS_LP_INFEAS),
     XPRM_CST_INT("MYXP_LP_CUTOFF",XPRS_LP_CUTOFF)
    };
```

6.2.3 List of types

The list of types has a single entry: a solver module needs to extend the Mosel type `mpproblem` with its own implementation.

```
static XPRMdsotyp tabtyp[]=
    {
     {"mpproblem.mxp",1,XPRM_DTYP_PROB|XPRM_DTYP_APPND,slv_pb_create,
      slv_pb_delete,NULL,NULL,slv_pb_copy}
    };
```

The following structure implements the *problem* type for our module, Mosel will maintain one instance of this type for each `mpproblem` object.

```
typedef struct SlvPb
    {
     struct SlvCtx *slctx;      /* Solver context */
     XPRSprob xpb;
     int have;
     int is_mip;
     double *solval;            /* Structures for storing solution values */
     double *dualval;
     double *rcostval;
     double *slackval;
     XPRMcontext saved_ctx;     /* Mosel context (used by callbacks) */
     struct SlvPb *prev,*next;
    } s_slvpb;
```

A *solver context* definition is shown below in Section 6.2.5.

6.2.4 List of services

The services `PARAM` and `PARLST` are required for the handling of module parameters, `RESET` and `UNLOAD` manage the access to the solver library.

```
static XPRMdsoserv tabserv[]=
    {
     {XPRM_SRV_PARAM, (void *)slv_findparam},
     {XPRM_SRV_PARLST, (void *)slv_nextparam},
     {XPRM_SRV_RESET, (void *)slv_reset},
     {XPRM_SRV_UNLOAD, (void *)slv_quitlib}
    };
```

6.2.5 Module context

The module context holds the type ID for the extended `mpproblem` type, module options, and a list of references to the problems that have been created by this module:

```
typedef struct SlvCtx           /* A context for this module */
    {
     int pbid;                  /* ID of type "mpproblem.mxp" */
     int options;               /* Runtime options */
     s_slvpb *probs;            /* List of created problems */
    } s_slvctx;
```

A specific interface structure required by the matrix generation is the following *MIP solver interface definition* that defines the shorthands to be used for identifying constraint and variable types and specifies the names of the functions for matrix generation, cleaning up solution information, and retrieving solution values for decision variables and constraints:

```
static mm_mipsolver xpress=
    {{'N','G','L','E','R','1','2'},
     {'+','I','B','P','S','R'},
     slv_loadmat,
     slv_clearsol,
     slv_getsol_v,
     slv_getsol_c};
```

6.2.6 Interface structure

The interface structure holds as usual the definition of the four tables (constants, subroutines, types, and services).

```
static XPRMdsointer dsointer=
    {
     sizeof(tabconst)/sizeof(XPRMdsoconst), tabconst,
     sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
     sizeof(tabtyp)/sizeof(XPRMdsotyp), tabtyp,
     sizeof(tabserv)/sizeof(XPRMdsoserv), tabserv
    };
```

6.2.7 Initialization function

The module initialization function performs the initialization of the solver library.

```
DSO_INIT myxprs_init(XPRMnifct nifct, int *interver,int *libver,
  XPRMdsointer **interf)
{
 int r;

 *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 module interface structure */
 r=XPRSinit(NULL);              /* Initialize the solver */
 if((r!=0)&&(r!=32))
 {
  nifct->dispmsg(NULL,"myxprs: I cannot initialize Xpress Optimizer.\n");
```

```
  return 1;
 }
 mm=nifct;                      /* Retrieve the Mosel NI function table */
 return 0;
}
```

6.3 Implementation of subroutines

6.3.1 Solver library calls

The first two entries of the list of subroutines concern the handling of module parameters, with the exception of `myxp_verbose` that is a setting for the module itself, all other parameters are straightforward mappings of solver control parameters published by the solver ibrary.

```
/**** Getting a control parameter ****/
static int slv_lc_getpar(XPRMcontext ctx,void *libctx)
{
 s_slvctx *slctx;
 int n;
 double r;

 slctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0:
   XPRM_PUSH_INT(ctx,(slctx->options&OPT_VERBOSE)?1:0);
   break;
  case 1:
   XPRSgetintcontrol(SLVCTX2PB(slctx)->xpb,XPRS_MAXTIME,&n);
   XPRM_PUSH_INT(ctx,n);
   break;
  case 2:
   XPRSgetintattrib(SLVCTX2PB(slctx)->xpb,XPRS_LPSTATUS,&n);
   XPRM_PUSH_INT(ctx,n);
   break;
  case 3:
   XPRSgetdblattrib(SLVCTX2PB(slctx)->xpb,XPRS_LPOBJVAL,&r);
   XPRM_PUSH_REAL(ctx,r);
   break;
  default:
   mm->dispmsg(ctx,"myxprs: Wrong control parameter number.\n");
   return XPRM_RT_ERROR;
 }
 return XPRM_RT_OK;
}

/**** Setting a control parameter ****/
static int slv_lc_setpar(XPRMcontext ctx,void *libctx)
{
 s_slvctx *slctx;
 int n;

 slctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0:
    slctx->options=XPRM_POP_INT(ctx)?(slctx->options|OPT_VERBOSE):(slctx->options&~OPT_VERBOSE);
    break;
  case 1:
    XPRSsetintcontrol(SLVCTX2PB(slctx)->xpb,XPRS_MAXTIME,XPRM_POP_INT(ctx));
    break;
  default:
    mm->dispmsg(ctx,"myxprs: Wrong control parameter number.\n");
```

```
    return XPRM_RT_ERROR;
 }

 return XPRM_RT_OK;
}
```

The subroutine `getprobstat` exposes the Mosel problem status at the model level (this status value is populated after every solver run, see implementation of function `slv_optim` below).

```
static int slv_lc_getpstat(XPRMcontext ctx,void *libctx)
{
 XPRM_PUSH_INT(ctx,mm->getprobstat(ctx)&XPRM_PBRES);
 return RT_OK;
}
```

The two module functions implementing the `minimize` and `maximize` subroutines map to the same function `slv_optim`.

```
static int slv_lc_maxim(XPRMcontext ctx,void *libctx)
{
 XPRMlinctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MAXIMIZE,obj);
}

static int slv_lc_minim(XPRMcontext ctx,void *libctx)
{
 XPRMlinctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MINIMIZE,obj);
}
```

The function `slv_optim` first clears any existing solution information, it then generates and loads the matrix representation of the problem into the solver and starts the actual solving process. After termination of the solver run it retrieves problem status information in order to populate Mosel's problem status flag.

```
static int slv_optim(XPRMcontext ctx, s_slvctx *slctx, int objsense, XPRMlinctr obj)
{
 int c,i;
 s_slvpb *slpb;
 int result;
 double objval;

 slpb=SLVCTX2PB(slctx);
 slpb->saved_ctx=ctx; /* Save current context for callbacks */
 slv_clearsol(ctx,slpb);

 /* Call NI function 'loadmat' to generate and load the matrix */
 if(mm->loadmat(ctx,obj,NULL,MM_MAT_FORCE,&xpress,slpb)!=0)
 {
  mm->dispmsg(ctx,"myxprs: loadprob failed.\n");
  slpb->saved_ctx=NULL;
  return RT_ERROR;
 }

 /* Set optimization direction */
 XPRSchgobjsense(slpb->xpb,
  (objsense==OBJ_MINIMIZE)?XPRS_OBJ_MINIMIZE:XPRS_OBJ_MAXIMIZE);

 mm->setprobstat(ctx,XPRM_PBSOL,0); /* Solution available for callbacks */
 if(!slpb->is_mip)
 {
```

```
 /* Solve an LP problem */
 c=XPRSlpoptimize(slpb->xpb,"");
 if(c!=0)
 {
  mm->dispmsg(ctx,"myxprs: optimisation failed.\n");
  slpb->saved_ctx=NULL;
  return RT_ERROR;
 }

 /* Retrieve solution status */
 XPRSgetintattrib(slpb->xpb,XPRS_PRESOLVESTATE,&i);
 if(i&128)
 {
  XPRSgetdblattrib(slpb->xpb,XPRS_LPOBJVAL,&objval);
  result=XPRM_PBSOL;
 }
 else
 {
  objval=0;
  result=0;
 }
 XPRSgetintattrib(slpb->xpb,XPRS_LPSTATUS,&i);
 switch(i)
 {
  case XPRS_LP_OPTIMAL:        result|=XPRM_PBOPT; break;
  case XPRS_LP_INFEAS:         result|=XPRM_PBINF; break;
  case XPRS_LP_CUTOFF:         result|=XPRM_PBOTH; break;
  case XPRS_LP_UNFINISHED:     result|=XPRM_PBUNF; break;
  case XPRS_LP_UNBOUNDED:      result|=XPRM_PBUNB; break;
  case XPRS_LP_CUTOFF_IN_DUAL: result|=XPRM_PBOTH; break;
  case XPRS_LP_UNSOLVED:       result|=XPRM_PBOTH; break;
 }
}
else
{
 /* Solve an MIP problem */
 c=XPRSmipoptimize(slpb->xpb,"");
 if(c!=0)
 {
  mm->dispmsg(ctx,"myxprs: optimization failed.\n");
  slpb->saved_ctx=NULL;
  return RT_ERROR;
 }

 /* Retrieve solution status */
 XPRSgetintattrib(slpb->xpb,XPRS_MIPSTATUS,&i);
 switch(i)
 {
  case XPRS_MIP_LP_NOT_OPTIMAL:
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_LP_OPTIMAL:
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_NO_SOL_FOUND: /* Search incomplete: no solution */
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_SOLUTION: /* Search incomplete: there is a solution */
     XPRSgetdblattrib(slpb->xpb,XPRS_MIPOBJVAL,&objval);
     result=XPRM_PBUNF|XPRM_PBSOL;
     slpb->have|=HAVEMIPSOL;
     break;
  case XPRS_MIP_INFEAS: /* Search complete: no solution */
     objval=0;
     result=XPRM_PBINF;
     break;
```

```
   case XPRS_MIP_OPTIMAL: /* Search complete: best solution available */
      XPRSgetdblattrib(slpb->xpb,XPRS_MIPOBJVAL,&objval);
      result=XPRM_PBSOL|XPRM_PBOPT;
      slpb->have|=HAVEMIPSOL;
      break;
   case XPRS_MIP_UNBOUNDED:
      objval=0;
      result=XPRM_PBUNB;
      break;
  }
  if(!(result&XPRM_PBSOL))
  {
   /* If no MIP solution try to get an LP solution */
   XPRSgetintattrib(slpb->xpb,XPRS_PRESOLVESTATE,&i);
   if(i&128)
   {
    XPRSgetdblattrib(slpb->xpb,XPRS_LPOBJVAL,&objval);
    result|=XPRM_PBSOL;
   }
  }
 }

 /* Record solution status and objective value */
 mm->setprobstat(ctx,result,objval);
 slpb->saved_ctx=NULL;
 return 0;
}
```

6.3.1.1 Implementation of MIP solver interface functions

The following functions implement the MIP solver interface functions ('loadmat', 'clearsol', 'getsol_v', 'getsol_c') that are communicated to the NI routine `loadmat` via the MIP solution interface structure `xpress` (for its definition see Section 6.2.5).

The 'loadmat' function loads a matrix that is held in Mosel structures into the LP or MIP solver.

```
static int slv_loadmat(XPRMcontext ctx, void *mipctx, mm_matrix *m)
{
 s_slvpb *slpb;
 s_slvctx *slctx;
 char pbname[80];
 int c,r;

 slpb=mipctx;
 slctx=slpb->slctx;
 slv_clearsol(ctx,slpb);
 slpb->is_mip=(m->ngents>0) || (m->nsos>0);

 sprintf(pbname,"xpb%p",slpb);
 if(slpb->is_mip)
  r=XPRSloadglobal(slpb->xpb,pbname,m->ncol,m->nrow,
m->qrtype,m->rhs,m->range,m->obj,
m->mstart,NULL,m->mrwind,m->dmatval,m->dlb,m->dub,
m->ngents,m->nsos,m->qgtype,m->mgcols,m->mplim,m->qstype,
m->msstart,m->mscols,m->dref);
 else
  r=XPRSloadlp(slpb->xpb,pbname,m->ncol,m->nrow,
m->qrtype,m->rhs,m->range,m->obj,
m->mstart,NULL,m->mrwind,m->dmatval,m->dlb,m->dub);

 /* Objective constant term */
 if(!r) { c=-1;r=XPRSchgobj(slpb->xpb,1, &c, &(m->fixobj)); }

 return r;
}
```

The 'clearsol' function frees up solution information held in the solver problem interface structures.

```
static void slv_clearsol(XPRMcontext ctx, void *mipctx)
{
 s_slvpb *slpb;

 slpb=mipctx;
 Free(&(slpb->solval));
 Free(&(slpb->dualval));
 Free(&(slpb->rcostval));
 Free(&(slpb->slackval));
 slpb->have=0;
}

/**** Free + reset memory ****/
static void Free(void *ad)
{
 free(*(void **)ad);
 *(void **)ad=NULL;
}
```

The 'getsol_v' and 'getsol_c' routines serve to retrieve the solution values for decision variables and constraints from the solver. These implementations retrieve the entire arrays at once and any subsequent calls return the information saved in the solver interface structures.

```
/**** Solution information for decision variables ****/
static double slv_getsol_v(XPRMcontext ctx, void *mipctx, int what, int col)
{
 s_slvpb *slpb;

 slpb=mipctx;
 if(what)
 {
  if(!(slpb->have&HAVERCS))
  {
   if(slpb->rcostval==NULL)
   {
    int ncol;

    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALCOLS,&ncol);
    if((slpb->rcostval=malloc(ncol*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
     return 0;
    }
   }
   if(slpb->have&HAVEMIPSOL) /* No rcost for a MIP => 0 */
   {
    int ncol;

    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALCOLS,&ncol);
    memset(slpb->rcostval,0,ncol*sizeof(double));
   }
   else
    XPRSgetlpsol(slpb->xpb,NULL,NULL,NULL,slpb->rcostval);
   slpb->have|=HAVERCS;
  }
  return slpb->rcostval[col];
 }
 else
 {
  if(!(slpb->have&HAVESOL))
  {
   if(slpb->solval==NULL)
   {
    int ncol;
```

```
    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALCOLS,&ncol);
    if((slpb->solval=malloc(ncol*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
     return 0;
    }
   }
   if(slpb->have&HAVEMIPSOL)
    XPRSgetmipsol(slpb->xpb,slpb->solval,NULL);
   else
    XPRSgetlpsol(slpb->xpb,slpb->solval,NULL,NULL,NULL);
   slpb->have|=HAVESOL;
  }
  return slpb->solval[col];
 }
}

/**** Solution information for linear constraints ****/
static double slv_getsol_c(XPRMcontext ctx, void *mipctx, int what, int row)
{
 s_slvpb *slpb;

 slpb=mipctx;
 if(what)
 {
  if(!(slpb->have&HAVEDUA))
  {
   if(slpb->dualval==NULL)
   {
    int nrow;

    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALROWS,&nrow);
    if((slpb->dualval=malloc(nrow*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
     return 0;
    }
   }
   if(slpb->have&HAVEMIPSOL) /* No dual for a MIP => 0 */
   {
    int nrow;

    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALROWS,&nrow);
    memset(slpb->dualval,0,nrow*sizeof(double));
   }
   else
    XPRSgetlpsol(slpb->xpb,NULL,NULL,slpb->dualval,NULL);
   slpb->have|=HAVEDUA;
  }
  return slpb->dualval[row];
 }
 else
 {
  if(!(slpb->have&HAVESLK))
  {
   if(slpb->slackval==NULL)
   {
    int nrow;

    XPRSgetintattrib(slpb->xpb,XPRS_ORIGINALROWS,&nrow);
    if((slpb->slackval=malloc(nrow*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
     return 0;
    }
   }
   if(slpb->have&HAVEMIPSOL)
    XPRSgetmipsol(slpb->xpb,NULL,slpb->slackval);
   else
```

```
    XPRSgetlpsol(slpb->xpb,NULL,slpb->slackval,NULL,NULL);
   slpb->have|=HAVESLK;
  }
  return slpb->slackval[row];
 }
}
```

6.3.2 Implementation of services

The RESET service function creates a new module context if none is provided in the argument, on a subsequent call where the module context argument is populated this context will be released after freeing all data structures that may have been created via the solver library.

```
/**** Reset the myxprs interface for a run ****/
static void *slv_reset(XPRMcontext ctx, void *libctx)
{
 s_slvctx *slctx;

 /* End of execution: release context */
 if(libctx!=NULL)
 {
  slctx=libctx;

  /* Release all remaining problems */
  while(slctx->probs!=NULL)
  {
   slv_pb_delete(ctx,slctx,slctx->probs,-1);
  }
  free(slctx);
  return NULL;
 }
 else
 {
  /* Begin of execution: create context */
  if((slctx=malloc(sizeof(s_slvctx)))==NULL)
  {
   mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
   return NULL;
  }
  memset(slctx,0,sizeof(s_slvctx));

  /* Record the problem ID of our problem type */
  mm->gettypeprop(ctx,mm->findtypecode(ctx,"mpproblem.mxp"),XPRM_TPROP_PBID,
   (XPRMalltypes*)&(slctx->pbid));
  return (void *)slctx;
 }
}
```

The UNLOAD service terminates the solver library (frees up the licence) that has been initialized from the module initialization.

```
/**** Called when unloading the library ****/
static void slv_quitlib(void)
{
 if(mm!=NULL)
 {
  XPRSfree();
  mm=NULL;
 }
}
```

The implementation of the module parameter access services PARAM and PARLST is similar to what we have seen for other modules (*e.g.* see Section 4.3).

```
/**** Find a control parameter ****/
```

```
static int slv_findparam(const char *name,int *type,int why,XPRMcontext ctx,
void *libctx)
{
 int n;

 for(n=0;n<SLV_NBPARAM;n++)
 {
  if(strcmp(name,myxprsparams[n].name)==0)
  {
   *type=myxprsparams[n].type;
   return n;
  }
 }
 return -1;
}

/**** Return the next parameter for enumeration ****/
static void *slv_nextparam(void *ref,const char **name,const char **desc,
int *type)
{
 long cst;

 cst=(long)ref;
 if((cst<0)||(cst>=SLV_NBPARAM))
  return NULL;
 else
 {
  *name=myxprsparams[cst].name;
  *type=myxprsparams[cst].type;
  *desc=NULL;
  return (void *)(cst+1);
 }
}
```

6.3.3 Handling optimization problems

Each Mosel model creates a default optimization problem of type `mpproblem` holding the constraints that are defined in the model. Further (sub)problems can be defined explicitly by the model developer, such as in the example shown at the beginning of this chapter (Section 6.1).

A solver module needs to implement an extension to the `mpproblem` type. Typically, the underlying data structure will include a reference to the solver problem representation, some status flags and structures to store solution information (see definition in Section 6.2.3 above). For our Xpress Optimizer example we have implemented the type handling routines to create, delete, and copy optimization problems. If the underlying solver can only handle a single problem, the implementation of the 'create' routine should prevent the creation of more than one problem and a 'copy' routine is most likely not required.

The following implementation of a problem creation routine for Xpress Optimizer creates the Optimizer problem, redirects the Optimizer output onto Mosel and it also defines some logging callbacks in order to intercept a program interruption.

```
/**** Create a new "problem" ****/
static void *slv_pb_create(XPRMcontext ctx, void *libctx, void *toref, int type)
{
 s_slvctx *slctx;
 s_slvpb *slpb;
 int i;

 slctx=libctx;
 if((slpb=malloc(sizeof(s_slvpb)))==NULL)
 {
  mm->dispmsg(ctx,"myxprs: Out of memory error.\n");
  return NULL;
 }
```

```
 memset(slpb,0,sizeof(s_slvpb));
 i=XPRScreateprob(&(slpb->xpb));
 if((i!=0)&&(i!=32))
 {
  mm->dispmsg(ctx,"myxprs: I cannot create the problem.\n");
  free(slpb);
  return NULL;
 }
 slpb->slctx=slctx;
 /* Redirect solver messages to the Mosel streams */
 XPRSaddcbmessage(slpb->xpb,slvcb_output,slpb,0);
 XPRSsetintcontrol(slpb->xpb,XPRS_OUTPUTLOG,1);

 /* Define log callbacks to report program interruption */
 XPRSaddcblplog(slpb->xpb,(void*)slvcb_stopxprs,slpb,0);
 XPRSaddcbcutlog(slpb->xpb,(void*)slvcb_stopxprs,slpb,0);
 XPRSaddcbgloballog(slpb->xpb,(void*)slvcb_stopxprs,slpb,0);
 XPRSaddcbbarlog(slpb->xpb,(void*)slvcb_stopxprs,slpb,0);

 if(slctx->probs!=NULL)
 {
  slpb->next=slctx->probs;
  slctx->probs->prev=slpb;
 }
 /* else we are creating the master problem */

 slctx->probs=slpb;
 return slpb;
}
```

The problem deletion routine needs to update the list of problems saved in the solver problem interface structure.

```
/**** Delete a "problem" ****/
static void slv_pb_delete(XPRMcontext ctx,void *libctx,void *todel,int type)
{
 s_slvctx *slctx;
 s_slvpb *slpb;

 slctx=libctx;
 slpb=todel;
 slv_clearsol(ctx,slpb);
 XPRSdestroyprob(slpb->xpb);
 if(slpb->next!=NULL) /* Last in list */
  slpb->next->prev=slpb->prev;
 if(slpb->prev==NULL) /* First in list */
  slctx->probs=slpb->next;
 else
  slpb->prev->next=slpb->next;
 free(slpb);
}
```

A problem is copied without duplicating the solutio information.

```
/**** Copy/reset/append problems: simply clear data of the destination ****/
static int slv_pb_copy(XPRMcontext ctx,void *libctx,void *toinit,void *src,int ust)
{
 s_slvpb *slpb;

 slpb=toinit;
 if(XPRM_CPY(ust)<XPRM_CPY_APPEND) slv_clearsol(ctx,slpb);
 return 0;
}
```

6.4 Implementing a solver callback

6.4.1 Example

Many programs, and in particular LP/MIP solvers, provide the possibility to interact with the program during its execution by means of *callbacks*. In terms of an example, we will show here how to implement an 'INTSOL' callback for Xpress Optimizer, that is, an etry point for calling a Mosel subroutine every time the solver has found a new MIP solution. The corresponding Mosel code might look as follows (notice that the Mosel subroutine is flagged as `public` in order to make it visible for external programs):

```
 public procedure intsol
   writeln("!!! New solution !!!")
  writeln("Solution: ", getobjval, "; ", x.sol, "; ", y.sol),
     "; obj=", getparam("myxp_lpobjval"))
 end-procedure

! Define the procedure 'intsol' as the solver INTSOL callback routine
 setcbintsol("intsol")
```

6.4.2 Implementation of callback handling

The handling of callbacks by the Mosel NI is not specific to the matrix / MIP solver interface, it can be applied for any external program that provides entry points for callbacks. In our case, the subroutine `setcbintsol` is declared via the following entry in the list of subroutines.

```
{"setcbintsol",2102,XPRM_TYP_NOT,1,"s",slv_lc_setcbintsol}
```

The implementation of the 'setcbintsol' routine in the module function `slvlc_setcbintsol` checks whether the specified subroutine has the expected format before saving its name in the problem structure.

```
static int slv_lc_setcbintsol(XPRMcontext ctx,void *libctx)
{
 s_slvctx *slctx;
 s_slvpb *slpb;
 XPRMalltypes result;
 const char *procname,*partyp;
 int nbpar,type;

 slctx=libctx;
 slpb=SLVCTX2PB(slctx);
 procname=XPRM_POP_REF(ctx);

 if(procname!=NULL)
 {                  /* The specified entity must be a procedure */
  if(XPRM_STR(mm->findident(ctx,procname,&result))!=XPRM_STR_PROC)
  {
   mm->dispmsg(ctx,"myxprs: Wrong subroutine type for callback 'intsol'.\n");
   return RT_ERROR;
  }
  do
  {                 /* The specified procedure must not have any arguments */
   mm->getprocinfo(result.proc,&partyp,&nbpar,&type);
   if((type==XPRM_TYP_NOT)&&(nbpar==0)) break;
   result.proc=mm->getnextproc(result.proc);
  } while(result.proc!=NULL);
  if(result.proc==NULL)
  {
   mm->dispmsg(ctx,"myxprs: Wrong procedure type for callback 'intsol'.\n");
   return RT_ERROR;
  }
```

```
  else
   slpb->cb_intsol=result.proc;
 }
 else
  slpb->cb_intsol=NULL;
 return RT_OK;
}
```

The problem structure `s_slvpb` has received a new field to store the callback reference:

```
 typedef struct SlvPb
{
 ...
 XPRMproc cb_intsol;
} s_slvpb;
```

We also add a line for initializing this callback to the problem creation routine `slv_pb_create` after the creation of the actual problem:

```
 /* Define intsol callback */
 XPRSaddcbintsol(slpb->xpb,(void*)slv_cb_intsol,slpb,0);
```

The callback function `slv_cb_intsol` needs to have the prototype required by the solver library. In addition to the invocation of the Mosel procedure specified in the model via `setcbintsol` we also add the handling of user interrupts to this implementation.

```
static void XPRS_CC slv_cb_intsol(XPRSprob opt_prob,s_slvpb *slpb)
{
 XPRMalltypes result;
 XPRSprob xpb_save;

 if(slpb->cb_intsol!=NULL)
 {
  xpb_save=slpb->xpb;
  slpb->xpb=opt_prob;
  slpb->have=0;
  if(mm->callproc(slpb->saved_ctx,slpb->cb_intsol,&result)!=0)
  {
   mm->stoprun(slpb->saved_ctx);
   XPRSinterrupt(opt_prob,XPRS_STOP_CTRLC);
  }
  slpb->xpb=xpb_save;
 }
}
```

6.5 Generating names for matrix entries

An LP/MIP problem definition held in Mosel can be displayed on screen or exported to a file using the subroutine `exportprob`. Nevertheless, in particular while developing a solver interface it may be helpful to also have the possibility of writing out the matrix representation directly from the solver.

In our example implementation the matrix gets loaded into the solver through the call to optimization, so writing out the matrix needs to take place after this call:

```
setparam("myxp_loadnames", true)
maximize(MyObj)
writeprob("mymat.lp", "l")
```

When writing out a matrix for debugging purposes one might expect to be able to match the rows and columns to the Mosel modeling entities via their respective names. By default, Mosel

does not generate any names for decision variables or constraints in order to maintain a low memory footprint. This feature needs to be added explicitly into the implementation of the `loadmat` routine that we have seen earlier in this chapter. After a call to the NI function `genmpnames` the resulting names are collected into the corresponding data structures that are expected by the solver library (uploading names for rows, columns, and SOS separately in the case of Xpress Optimizer).

```
static int slv_loadmat(XPRMcontext ctx,void *mipctx,mm_matrix *m)
{
 s_slvpb *slpb;
 s_slvctx *slctx;
 int c,r;

 slpb=mipctx;
 slctx=slpb->slctx;

 /* Generate names for matrix elements */
 if(slctx->options&OPT_LOADNAMES)
  mm->genmpnames(ctx,MM_KEEPOBJ,NULL,0);

 /* ... load the problem matrix into the solver ... */

 /* Load names if requested */
 if(!r && (slctx->options&OPT_LOADNAMES))
 {
  char *names,*n;
  size_t totlen,totlen2;
  size_t l;

  totlen=0;
  for(c=0;c<m->ncol;c++)
  {
   l=strlen(mm->getmpname(ctx,MM_MPNAM_COL,c));
   totlen+=l+1;
  }
  totlen2=0;
  for(c=0;c<m->nrow;c++)
  {
   l=strlen(mm->getmpname(ctx,MM_MPNAM_ROW,c));
   totlen2+=l+1;
  }
  if(totlen<totlen2) totlen=totlen2;
  totlen2=0;
  for(c=0;c<m->nsos;c++)
  {
   l=strlen(mm->getmpname(ctx,MM_MPNAM_SOS,c));
   totlen2+=l+1;
  }
  if(totlen<totlen2) totlen=totlen2;

  if((names=malloc(totlen))==NULL)
   mm->dispmsg(ctx,"myxprs: Not enough memory for loading the names.\n");
  else
  {
   n=names;
   for(c=0;c<m->ncol;c++)
    n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_COL,c)))+1;
   if((r=XPRSaddnames(slpb->xpb,2,names,0,m->ncol-1))!=0)
     mm->dispmsg(ctx,"myxprs: Error when executing ‘addnames’.\n");
   if(!r && (m->nrow>0))
   {
    n=names;
    for(c=0;c<m->nrow;c++)
     n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_ROW,c)))+1;
    if((r=XPRSaddnames(slpb->xpb,1,names,0,m->nrow-1))!=0)
     mm->dispmsg(ctx,"myxprs: Error when executing ‘addnames’.\n");
   }
   if(!r && (m->nsos>0))
```

```
   {
    n=names;
    for(c=0;c<m->nsos;c++)
     n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_SOS,c)))+1;
    if((r=XPRSaddnames(slpb->xpb,3,names,0,m->nsos-1))!=0)
     mm->dispmsg(ctx,"myxprs: Error when executing ‘addnames’.\n");
   }
   free(names);
  }
 }
 return r;
}
```

In this subroutine, the loading of names is subject to the presence of the option flag `LOADNAMES` that is set via a new module parameter `myxp_loadnames` which is declared via the following entry in the table of parameters:

```
{"myxp_loadnames",XPRM_TYP_BOOL|XPRM_CPAR_READ|XPRM_CPAR_WRITE}
```

6.5.1 Implementing the ’writeprob’ subroutine

The `writeprob` routine shown in the Mosel model extract at the beginning of this section needs to be declared in the table of subroutines structure by adding the following line to it:

```
 {"writeprob",2103,XPRM_TYP_NOT,2,"ss",slvlc_writepb}
```

And the actual implementation in the function `slvlc_writepb` consists of a call the the solver’s matrix output function, along with some error handling such as a check for write access to the specified location:

```
static int slvlc_writepb(XPRMcontext ctx,void *libctx)
{
 s_slvpb *slpb;
 int rts;
 char *dname,*options;
 char ename[MM_MAXPATHLEN];

 slpb=SLCTX2PB((s_slvctx*)libctx);
 dname=MM_POP_REF(ctx);
 options=MM_POP_REF(ctx);
 if((dname!=NULL)&&             /* Make sure the file can be created */
    (mm->pathcheck(ctx,dname,ename,MM_MAXPATHLEN,MM_RCHK_WRITE|MM_RCHK_IODRV)==0))
 {
  slpb->saved_ctx=ctx;         /* Save current context for callbacks */
  rts=XPRSwriteprob(slpb->xpb,XNLSconvstrto(XNLS_ENC_FNAME,ename,-1,NULL),options);
  slpb->saved_ctx=NULL;
  if(rts)
  {
   mm->dispmsg(ctx,"myxprs: Error when executing ‘writeprob’.\n");
   return RT_IOERR;
  }
  else
   return RT_OK;
 }
 else
 {
  mm->dispmsg(ctx,"myxprs: Cannot write to ’%s’.\n",dname!=NULL?dname:"");
  return RT_IOERR;
 }
}
```

CHAPTER 7

Defining a static module

Modules are libraries that provide additional functionality for the Mosel language. They are usually created as dynamic shared objects that can be used independently of the way a Mosel program is executed. If however, a Mosel program is compiled and run from within a C program (using the Mosel libraries), it is possible to include the definition of a module used by the Mosel program into the C program, thus creating a *static module*. Such a static module is only visible to and usable by Mosel programs that are executed from this C program. (The C file is compiled into a standard object file, no `.dso` file is created for the module.)

This chapter gives an example of a typical use of such a static module: for a Mosel program that is embedded into some large application it certainly is preferable to load data already held in memory directly into the model structures and not having to pass them via data files.

7.1 Example

We would like to initialize an array of integers in a Mosel program with data held in the C program that executes it:

```
model "Test initialization in memory"
 uses "meminit"

 parameters
  MEMDAT=''              ! Location of data in memory
  MEMSIZ=0               ! Size of the data block (nb of integers)
 end-parameters

 declarations
  a:array(1..20) of integer
 end-declarations

 writeln("Data located at ", MEMDAT, " contains ", MEMSIZ, " integers")
 meminit(a, MEMDAT, MEMSIZ)
 writeln("a=", a)

end-model
```

A C program to execute the Mosel program `meminit_test.mos` printed above may look as follows:

```
int main()
{
 XPRMmodel mod;
 int result;
 char params[80];
 static int tabinit[]= {23,78,45,90,234,111,900,68,110};

 XPRMinit();                                      /* Initialize Mosel */
 XPRMcompmod("", "meminit_test.mos", NULL, NULL);  /* Compile the model */
```

```
 mod=XPRMloadmod("meminit_test.bim", NULL);        /* Load the model */

 /* Parameters: the address of the data table and its size */
 sprintf(params, "MEMDAT='%p', MEMSIZ=%d", tabinit, sizeof(tabinit)/sizeof(int));

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

7.2 Structures for passing information

A static module differs from dynamic modules only in the way it is initialized. The module initialization function (see below Section 7.2.2) has no special return type to make it known to Mosel, instead it is declared to Mosel in the main C program. After the initialization of Mosel, but before any model file that uses the static module *meminit* is compiled or loaded, we have to add the following line:

```
XPRMregstatdso("meminit", meminit_init);
```

The function `XPRMregstatdso` registers the module name and its initialization function with Mosel.

7.2.1 List of subroutines

The module *meminit* only defines a single subroutine, namely the procedure `meminit`. This procedure takes three arguments (see Appendix A.2.2 for an explanation of the encoding of the parameter format string): `AI.i`: an array of integers indexed by a range (the data we want to pass to the model), `s`: a string (the location of the data in memory) and `i`: an integer (the size of the data array):

```
static XPRMdsofct tabfct[]=
    {
     {"meminit", 1000, XPRM_TYP_NOT, 3, "AI.isi", mi_meminit}
    };
```

This table of functions needs to be included into the main interface structure as shown in the previous chapters.

7.2.2 Initialization function

As mentioned earlier, the prototype of the initialization function for static modules is slightly different from what we have seen for DSOs, but the information exchanged between Mosel and the module is the same:

```
static int meminit_init(XPRMnifct nifct, int *interver, int *libver,
                        XPRMdsointer **interf)
{
 mm=nifct;                      /* Save the table of 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;
}
```

7.3 Complete module example

Below follows the complete code of the static module *meminit* and the main function that declares this module and executes the Mosel model which requires the module.

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

static int meminit_init(XPRMnifct nifct, int *interver, int *libver,
                        XPRMdsointer **interf);

/* Main function */
int main()
{
 XPRMmodel mod;
 int result;
 char params[80];
 static int tabinit[]= {23,78,45,90,234,111,900,68,110};

 XPRMinit();                                        /* Initialize Mosel */

 /* Register 'meminit' as a static module (=stored in the program) */
 XPRMregstatdso("meminit", meminit_init);

 XPRMcompmod("", "meminit_test.mos", NULL, NULL);  /* Compile the model */
 mod=XPRMloadmod("meminit_test.bim", NULL);        /* Load the model */

 /* Parameters: the address of the data table and its size */
 sprintf(params, "MEMDAT='%p', MEMSIZ=%d", tabinit, sizeof(tabinit)/sizeof(int));

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

/ ******************** **** Body of the module 'meminit'  ******************** ****/

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

/* List of subroutines */
static XPRMdsofct tabfct[]=
    {
     {"meminit", 1000, XPRM_TYP_NOT, 3, "AI.isi", mi_meminit}
    };

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

static XPRMnifct mm;            /* To store the mosel function table */

/* Initialization function of the module */
static int meminit_init(XPRMnifct nifct, int *interver, int *libver,
                        XPRMdsointer **interf)
{
 mm=nifct;                      /* Save the table of 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;
}
```

```
/* Implementation of procedure 'meminit' */
static int mi_meminit(XPRMcontext ctx, void *libctx)
{
 XPRMarray arr;
 XPRMstring adr_s;
 XPRMset ndxset;
 int *adr,siz,index[1],last,i;

 arr=XPRM_POP_REF(ctx);             /* The array */
 adr_s=XPRM_POP_STRING(ctx);        /* Data location (as a string) */
 siz=XPRM_POP_INT(ctx);             /* Data size */
 sscanf(adr_s,"%p",&adr);           /* Get the address from the string */

 mm->getarrsets(arr,&ndxset);
 index[0]=mm->getfirstsetndx(ndxset);
 last=mm->getlastsetndx(ndxset);
 for(i=0;(i<siz) && (index[0]<=last);i++,index[0]++)
  mm->setarrvalint(ctx,arr,index,adr[i]);
 return XPRM_RT_OK;
}
```

7.4 Turning a static module into a DSO

It requires only little work to transform a static module into a dynamic one (and vice versa). Assuming we would like to turn our module *meminit* into a DSO, we simply have to

- save all the functions of the module and the definition of the structures for passing information into a separate file;
- replace the prototype of the module initialization function by the following:

```
DSO_INIT meminit_init(XPRMnifct nifct, int *interver, int *libver,
                      XPRMdsointer **interf)
```

7.5 Static modules versus I/O drivers

The generalization of the notion 'file' and the introduction of I/O drivers in Mosel replace certain uses of static user modules. In particular for transfering data in memory it is often no longer necessary to write a dedicated module. However, other uses of static modules persist, such as the compilation of a standard module as a static module for debugging purposes.

The example from Section 7.1 may be re-written as follows using the `raw` and `mem` drivers that are available with the standard distribution of Mosel:

```
model "Test initialization in memory (I/O)"
 parameters
  MEMDAT=''              ! Data block in memory
 end-parameters

 declarations
  a:array(1..20) of integer
 end-declarations

 initializations from "raw:"
  a as MEMDAT
 end-initializations

 writeln("a=", a)
end-model
```

The *complete* C program to execute the Mosel program `meminitio.mos` printed above may look as follows:

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

int main()
{
 XPRMmodel mod;
 int result;
 char params[80];
 static int tabinit[]= {23,78,45,90,234,111,900,68,110};

 XPRMinit();                                    /* Initialize Mosel */
 XPRMcompmod("", "meminitio.mos", NULL, NULL);  /* Compile the model */
 mod=XPRMloadmod("meminitio.bim", NULL);        /* Load the model */

 /* Parameters: the address of the data table and its size */
 sprintf(params, "MEMDAT='noindex,mem:%p/%u'", tabinit, sizeof(tabinit));

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

CHAPTER 8

Compatibility checks: Handling versions and restrictions

The Mosel Native Interface, any modules using the NI, and also Mosel models using module functionality all are likely to evolve over time—most often via the addition of new functionality. In a development context, the components of an application typically are compiled and run using the same Mosel version. However, this may not be the case for deployment, resulting in issues of version compatibility, in particular when deploying (partial) updates to older platforms. The following sections show how such backwards compatibility can be achieved for Mosel modules.

Furthermore, the use of Mosel in protected environments, usually in the context of remote model execution (*e.g.*, via *mmjobs*, XPRD, Optimization Modeler), requires modules to be compliant with access restrictions that are imposed by the environment. Section 8.3 shows how to implement the necessary checks.

8.1 Mosel version

By default, Mosel modules will require (at least) the Mosel version that has been used for compiling them. That is, they cannot be run on older versions of Mosel. However, if we know that a module is not using any recent features of Mosel (*e.g.* it has been developed some time ago using an older version of Mosel and we are now simply recompiling it with some newer release) we can set the Mosel NI compatibility flag `XPRM_NICOMPAT` to some older version number.

```
#define XPRM_NICOMPAT 3002000          /* Compatibility level: Mosel 3.2.0 */
```

8.2 Module version

Modules that are developed and distributed over a longer period of time most likely will have gone through a number of versions— the reader is reminded that the module version number is returned in the argument `libver` of the module initialization function and can be generated with the help of the NI macro `XPRM_MKVER`. Functionality added by a given module version will obviously not be available from older versions, and inversely, models written for older module versions will not require this new functionality. So, depending on the functionality used by a given model, we may be required to use a more or less recent version of a given DSO (NB: the expected module version is stored in the BIM file).

Furthermore, depending on the conventions used for numbering a particular module, the default module version compatibility rules applied by Mosel may have to be modified for a particular module.

8.2.1 'Update version' service

The service XPRM_SRV_UPDVERS makes it possible to determine which module version is required by a model by inspecting the module functionality used in this particular model. This services is used by the Mosel compiler (whereas most other services are used at runtime). The DSO will be loaded with the lowest version number that satisfies the functionality required be the model. As a consequence, it is possible update the DSO to some newer version (containing additional functionality and maintaining all existing) without having to recompile the model source.

Example: Assume that we have implemented a new version 0.0.2 of the example 'solarray' from Chapter 2 that defines an additional overloaded form of the subroutine 'getsol' returning solution values rounded to integers. That is, we now have the following two entries in the list of subroutines:

```
static XPRMdsofct tabfct[]=
    {
     {"solarray",1000,XPRM_TYP_NOT,2,"A.vA.r",ar_getsol},
     {"solarray",1001,XPRM_TYP_NOT,2,"A.vA.i",ar_getintsol}
    };
```

To implement detailed module version checking, we define the list of services with a single entry for the XPRM_SRV_UPDVERS service, and we update the main interface definition structure correspondingly:

```
/* Table of services */
static XPRMdsoserv tabserv[]=
    {
     {XPRM_SRV_UPDVERS,(void*)updvers},      /* Module version check */
    };

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

The function updvers returns the required module version depending on the input it receives in its arguments event (XPRM_UPDV_INIT = called at module initialization, returns the lowest version counter for this module; XPRM_UPDV_FUNC = checking the module version required by a particular subroutine) and what (identification number of the object for subroutines, types, parameters).

```
static void updvers(int event, int what, int *version)
{
 if (event==XPRM_UPDV_INIT)
   *version=XPRM_MKVER(0,0,1);               /* First version of this module */
 else if (event==XPRM_UPDV_FUNC)
 {
   switch (what) {
    case 1000: *version=XPRM_MKVER(0,0,1);   /* Works with 1st module version */
               break;
    case 1001: *version=XPRM_MKVER(0,0,2);   /* Requires 2nd module version */
   }
 }
}
```

A model that is compiled using the new module version 0.0.2, but that just uses the original real-valued getsol function will load the module as version 0.0.1.

8.2.2 'Check version' service

The service XPRM_SRV_CHKVER allows a module to override Mosel's *default version compatibility rules* that are checked at runtime when loading the module: module version numbers use a code with 3 numbers (*major, minor, release*); by default, a module version *A* can be used in place of module version *B* if the following conditions apply

$$major(A) = major(B)$$
$$minor(A) = minor(B)$$
$$release(A) \geq release(B)$$

Example: Instead of numbering our extension of the 'solarray' example described in the previous section as version 0.0.2, we wish to give it the version number 0.1.0. That is, we now have the following module initialization function:

```
DSO_INIT solarray_init(XPRMnifct nifct, int *interver,int *libver, XPRMdsointer **interf)
{
 ...
 *libver=XPRM_MKVER(0,1,0);             /* Module version */
 ...
}
```

The compilation of models requiring the previous version 0.0.1 will work correctly with the corresponding definition of the 'update version' service. However, loading of the generated BIM file will fail with the error message 'wrong version for module solarray' if the default compatibility rules are applied. The following definition of the XPRM_SRV_CHKVER service will make it possible to use a module version 0.1.0 with a model that expects a DSO version 0.0.*, for completeness' sake we also show the definition of function updvers:

```
/* Table of services */
static XPRMdsoserv tabserv[]=
    {
     {XPRM_SRV_UPDVERS,(void*)updvers},
     {XPRM_SRV_CHKVER,(void*)chkvers},
    };

static void updvers(int event, int what, int *version)
{
 if (event==XPRM_UPDV_INIT)
   *version=XPRM_MKVER(0,0,1);               /* First version of this module */
 else if (event==XPRM_UPDV_FUNC)
 {
   switch (what) {
    case 1000: *version=XPRM_MKVER(0,0,1);   /* Works with 1st module version */
               break;
    case 1001: *version=XPRM_MKVER(0,1,0);   /* Requires 2nd module version */
   }
 }
}

static int chkvers(int reqvers)
{
 /* This module version accepts to run with models expecting any version
    from 0.0.1 to 0.1.0 inclusive */
  return (reqvers>MM_MKVER(0,1,0))||(reqvers<MM_MKVER(0,0,1));
}
```

8.3 Restrictions

Restrictions are implemented via the service XPRM_SRV_CHRES that expects a function of the form

`int chkres(int restr)`, where the restrictions to be checked are passed in the bit-encoded parameter `restr`.

```
static int chkres(int);

static XPRMdsoserv tabserv[]=
    {
     {XPRM_SRV_CHKRES,(void*)chkres},
    };
```

None of the example modules presented in this guide involve external file access or calls to external commands and therefore do not require any detailed checks of access restrictions. In all examples, we can simply add a function `chkres` that returns the value '0' to indicate that the module is compliant with all access restrictions, without any need for further modifications to the module definitions.

```
static int chkres(int r)
{
 return 0;
}
```

An example module that employs external file access functions is the data compression module *zlib* described in the whitepaper 'Generalized file handling in Mosel'. In this example, the 'CHKRES' service and function `chkres` are defined as shown above and in addition, we need to check for restrictions wherever external file access functions are being used, such as calls to file opening through an external compression library.

The relevant part of the 'open a compressed file' function `gzip_open` is shown below (the complete code of this example is provided in the file `zlib.c`, located in the directory `examples/mosel/Modules` of the Xpress distribution). Notice the use of the NI function `pathcheck` to expand the file name and check whether it can be accessed given the current restrictions. NB: Mosel NI file access functions such as `fopen` automatically perform the necessary tests for restrictions and hence do not require the addition of any tests to achieve compliance with restrictions.

```
static void *gzip_open(XPRMcontext ctx,int *mode,const char *fname)
{
 char cfname[MM_MAXPATHLEN];
 char cmode[16];

 if((fname==NULL)||
    (mm->pathcheck(ctx,fname,cfname,MM_MAXPATHLEN,
    ((*mode)&MM_F_WRITE)?MM_RCHK_WRITE:MM_RCHK_READ)!=0))
 {
  errno=EACCES;
  return NULL;
 }
 else
 {
  ...                                /* Build up 'cmode' */
  return gzopen(cfname,cmode);       /* Call external file access function */
 }
}
```

Appendix

APPENDIX A

Interface structures and function prototypes

This appendix lists the five structures for passing information from modules to Mosel together with the available options, macro definitions and predefined function prototypes that are used in this manual.

For a complete list and more detailed explanations see the Mosel Native Interface Reference Manual.

A.1 Module initialization

The module initialization function takes the following form. It must always be present. The function name must correspond to the name of the module, with `_init` appended to it. The first parameter passes the list of Mosel NI functions to the module, the other three parameters must be filled in by the module. The initialization function returns 0 if executed successfully, 1 otherwise.

```
DSO_INIT  modulename_init(XPRMnifct nifct,
                          int *interver,
                          int *libver,
                          XPRMdsointer **interf)
```

Arguments:

`nifct`	List of Native Interface functions provided by Mosel
`interver`	Native Interface version used by the module, must be set to `XPRM_NIVERS`
`libver`	Module version. The macro `XPRM_MKVER` can be used to compose a version number of three integers, for example with `XPRM_MKVER(0,0,1)` the smallest possible value (namely 0.0.1) is obtained
`interf`	Interface structure

A.2 Structures for passing information

The main *interface structure* that must be passed to Mosel in the module initialization function holds the lists of constants, subroutines, types and services that are provided by the module. Each list is preceded by an integer value that indicates its size. A list and its size may be NULL and 0 respectively if the module does not define any object of the corresponding category.

Structure `XPRMdsointer`:

```
{
 int sizec; XPRMdsoconst *tabconst;
 int sizef; XPRMdsofct *tabfct;
 int sizet; XPRMdsotyp *tabtyp;
 int sizes; XPRMdsoserv *tabserv;
};
```

A.2.1 List of constants

Structure `XPRMdsoconst`:

```
{
  constant_definition
};
```

A *constant_definition* contains the name of a constant, its type and its value. It is best obtained through one of the following macros:

```
XPRM_CST_INT(char *name, int value)
XPRM_CST_BOOL(char *name, int value)
XPRM_CST_STRING(char *name, char *value)
XPRM_CST_REAL(char *name, static const double value)
```

Note that the value of real constants cannot be set directly in this list but must be given via a C variable of type `static const double`.

A.2.2 List of subroutines

Structure `XPRMdsofct`:

```
{
 char *name;
 int code;
 int type;
 int nbpar;
 char *typpar;
 int (*vimfct)(XPRMcontext ctx, void *libctx);
}
```

The entries of this structure need to be defined as follows:

name name of the subroutine, or operator sign preceded by '@'; empty string for `getparam` and `setparam`. It is not possible to use any reserved word (the complete list is given in the Mosel Reference Manual) as the name of a subroutine.

code reference number for the type within the module. It must not be smaller than 1000 and be given in ascending order; value `XPRM_FCT_GETPAR` for function `getparam` (must be first in the list) and `XPRM_FCT_SETPAR` for procedure `setparam` (must come second) if these are defined by the module.

type type of the return value.

`XPRM_TYP_NOT`	no return value (procedure)
`XPRM_TYP_INT`	integer
`XPRM_TYP_REAL`	real number
`XPRM_TYP_STRING`	text string
`XPRM_TYP_BOOL`	Boolean
`XPRM_TYP_EXTN`	external type defined by this module (the exact type must be indicated in the parameter format string `typpar`)

nbpar number of parameters.

typpar string with parameter types (in the order of their appearance in the subroutine) or operand types. If the return value is an external type the string starts with the name of the type, separating it with a colon from the parameter format.

`i`	an integer
`r`	a real
`s`	a text string
`b`	a Boolean
`v`	a decision variable (type `mpvar`)
`c`	a linear constraint (type `linctr`)
`I`	a range set
`a`	an array (of any kind)
`e`	a set (of any type)
`|`*xxx*`|`	xternal type named '*xxx*'
`!`*xxx*`!`	the set named '*xxx*'
`A`*ndx*`.`*t*	an array indexed by '*ndx*' of the type '*t*'. '*ndx*' is a string describing the type of each indexing set. '*ndx*' may be omitted in which case any array of type '*t*' is a valid parameter.
`E`*t*	a set of type '*t*'
`L`*t*	a list of type '*t*'
`*`	must be the last character to indicate that the function has a variable number of arguments

vimfct the module library function that implements this subroutine or operator. The first argument is the context of Mosel (type `XPRMcontext`), the second the context of the module. For return codes see Section A.4.

A.2.2.1 Overview on operators in Mosel

All operators have a two-character name, the first character of which is always '@'. Operators can be defined for any type and return any type, however, they cannot replace a predefined operator. For instance the addition of reals @+(r,r):r cannot be re-defined in a module.

Typically, only a subset of all possible operators needs to be defined for a given type. For instance, arithmetic and logical operators are usually not applied to the same objects. Furthermore, in certain cases Mosel is able to deduce the definition of an operator (and also of aggregate operators) if some other operators are defined, so that it is not necessary to define all operators. Any implications that may be drawn are noted in the following list. Where operations are marked 'commutative', Mosel deduces the result for (B,A) if the operation is defined for (A,B), assuming that A and B are of different types.

In the following list (Table A.1), read $\rightarrow$ as 'returns'.

Table A.1: Overview on operators

Operator	Operation	Return value	Remarks
Basic constructors			
@&(C):C	duplication (cloning)	new object	
@&(params):C	construction	new object	
@0:C	identity for sums (0-element)	new object	implies aggregate `SUM` if @+(C,C):C defined and aggregate `OR` if @o(C,C):C defined
@1:C	identity for products (1-element)	new object	implies aggregate `PROD` if @*(C,C):C defined and aggregate `AND` if @a(C,C):C defined
Assignment operators			
@:(C,A)	direct assignment	C:=A	
@M(C,A)	subtractive assignment	C-=A	implied by @:(C,A) with @-(C,A):C
@P(C,A)	additive assignment	C+=A	implied by @:(C,A) with @+(C,A):C
Arithmetic operators			
@+(A,B):C	addition	A + B $\rightarrow$ C	commutative
@-(A,B):C	subtraction	A - B $\rightarrow$ C	implied by @+(A,B):C with @-(B):B
@-(A):C	negation	- A $\rightarrow$ C	
@*(A,B):C	multiplication	A * B $\rightarrow$ C	commutative
@/(A,B):C	division	A / B $\rightarrow$ C	
@d(A,B):C	integer division	A div B $\rightarrow$ C	
@m(A,B):C	modulo operation	A mod B $\rightarrow$ C	
@^(A,B):C	exponential operation	$A^B \rightarrow C$	
Logical operators			
@a(A,B):C	logical 'and'	A and B $\rightarrow$ C	
@o(A,B):C	logical 'or'	A or B $\rightarrow$ C	
@n(A):C	logical negation	not A $\rightarrow$ C	
Comparators			
@<(A,B):C	strictly less	A<B $\rightarrow$ C	implied by @n(C):C with @g(A,B):C
@>(A,B):C	strictly greater	A>B $\rightarrow$ C	implied by @n(C):C with @l(A,B):C
@l(A,B):C	less or equal	A $\leq$ B $\rightarrow$ C	implied by @n(C):C with @>(A,B):C
@g(A,B):C	greater or equal	A $\geq$ B $\rightarrow$ C	implied by @n(C):C with @<(A,B):C
@=(A,B):C	equality	A=B $\rightarrow$ C	implied by @n(C):C with @#(A,B):C, commutative
@#(A,B):C	difference	A $\neq$ B $\rightarrow$ C	implied by @n(C):C with @=(A,B):C
is_ operators			
@*e*(B):C	SOS type 1	B is_sos1 $\rightarrow$ C	
@*t*(B):C	SOS type 2	B is_sos2 $\rightarrow$ C	
@*f*(A):C	free	A is_free $\rightarrow$ C	
@*c*(A):C	continuous	A is_continuous $\rightarrow$ C	
@*i*(A):C	integer	A is_integer $\rightarrow$ C	
@*b*(A):C	binary	A is_binary $\rightarrow$ C	
@*p*(A,B):C	partial integer	A is_partint B $\rightarrow$ C	
@*s*(A,B):C	semi continuous	A is_semcont B $\rightarrow$ C	
@*r*(A,B):C	semi continuous integer	A is_semint B $\rightarrow$ C	
@_(A)	expression A is accepted as statement		

If A and B are of external types, they must be deleted by the operator with the exception of comparators where nothing is to be deleted.

The arguments of a subroutine or the objects that an operator is applied to must be obtained from the *stack* in the order that is specified in the format string `typpar` (see Section A.3, macros for taking objects from the stack). If the library function implements a function (that is, if argument `type` has a value other than `XPRM_TYP_NOT`), the value that is to be returned by the function must be put back onto the stack (see Section A.3, macros for putting objects onto the

stack).

A.2.3 List of types

Structure `XPRMdsotyp`:

```
{
 char *name;
 int code;
 int props;
 void *(*create)(XPRMcontext ctx, void *libctx, void *ref, int typnum);
 void (*fdelete)(XPRMcontext ctx, void *libctx, void *todel, int typnum);
 int (*tostring)(XPRMcontext ctx, void *libctx, void *toprt, char *dest,
                 int maxsize, int typnum);
 int (*fromstring)(XPRMcontext ctx, void *libctx, void *toinit,
                 const char *src, int typnum, const char **end);
 void (*copy)(XPRMcontext ctx, void *libctx, void *dest, void *src, int tnop);
 void (*compare)(XPRMcontext ctx, void *libctx, void *t1, void *t2, int tnop);
}
```

The entries of this structure have the following meaning (see the Mosl NI Reference Manual for details):

name name of the type. It is not possible to use any reserved word (the complete list is given in the Mosel Reference Manual) as the name of a type.

code reference number for the type within the module, must be smaller then 65536 and listed in ascending order.

props bit coded set of properties.

create type creation function (required).

fdelete type deletion function (NULL if none defined).

tostring function for converting type to a string (NULL if none defined).

fromstring function for initializing type from a string (NULL if none defined).

copy type copy function (NULL if none defined).

compare type compare function (NULL if none defined).

A.2.4 List of services

Structure `XPRMdsoserv`:

```
{
 int code;
 void *ptr;
}
```

The `code` indicates the type of service that is provided by the function (or data structure) `ptr`. The format of the pointer `ptr` depends on the service that it provides:

`XPRM_SRV_PARAM` Encode a parameter: for a given parameter name, this function fills in the type information and returns the reference number if it is defined in the module, otherwise it returns -1. This function must be provided if the module defines any control parameters.
`int findparam(const char *name, int *type)`

`XPRM_SRV_PARLST` Enumerate the parameter names. Mosel calls this function repeatedly until it returns NULL. At its first execution, the value of `ref` is NULL, at any subsequent call, it contains the value that has been returned by the preceding function call. The definition of this function is optional. Only if it is defined does the command `examine` of the Mosel Command Line Interpreter display the list of parameters provided by a module.

```
void *nextparam(void *ref, const char **name,
     const char **desc, int *type)
```

`XPRM_SRV_RESET` Reset a DSO for a run. This function is called at the start and termination of the execution of a Mosel program that uses the module. It should be used to create/initialize and, at the second call, to delete any internal structures of the module (its context) that need to be kept in memory during the execution of a Mosel program. Among others, the definition of new types requires this service.

```
void *reset(XPRMcontext ctx, void *libctx, int version)
```

The complete set of services provided by the Mosel Native Interface is documented in the NI Reference Manual. In addition to the service functions listed above, there are also services to override the default module version control, to handle licencing of modules, to enable inter-module communication, to indicate dependencies on other modules, and to define the I/O drivers implemented by a module.

A.2.5 Parameters

It may be convenient to store the control parameters provided by a module in a structure similar to the following:

```
struct
{
 char *name;
 int type;
 char *desc;
}
```

where `name` is the parameter name (it must always be given in lower case), `type` the type and access rights, and `desc` an optional description of the parameter that is displayed with the command `examine` of the Mosel Command Line Interpreter if the `PARLST` service is defined for the module. The type encoding will be composed of the parameter type that is one of

`XPRM_TYP_INT` — an integer number
`XPRM_TYP_REAL` — a real number
`XPRM_TYP_STRING` — a text string
`XPRM_TYP_BOOL` — a Boolean

and the read/write flags (if a flag is not set, the feature is disabled):

`XPRM_CPAR_READ` — read-enabled
`XPRM_CPAR_WRITE` — write-enabled

For example

```
XPRM_TYP_REAL|XPRM_CPAR_READ|XPRM_CPAR_WRITE
```

defines a real-valued parameter that is read-write-enabled.

A.3 Working with the stack

The Native Interface provides two sets of macros for accessing the stack: ‘pop’ and ‘push’. These macros must be used in order to obtain the values of the arguments for subroutines and parameters and to return the results of functions to Mosel.

Macros for taking objects from the stack:

```
XPRM_POP_INT(XPRMcontext ctx)
XPRM_POP_REAL(XPRMcontext ctx)
XPRM_POP_STRING(XPRMcontext ctx)
XPRM_POP_REF(XPRMcontext ctx)
```

Macros for putting objects onto the stack:

```
XPRM_PUSH_INT(XPRMcontext ctx, i)
XPRM_PUSH_REAL(XPRMcontext ctx, r)
XPRM_PUSH_STRING(XPRMcontext ctx, s)
XPRM_PUSH_REF(XPRMcontext ctx, r)
```

Only the basic types integer, real and string are passed directly to and from the stack. Boolean values are treated as integers. All other types are passed by reference (macros `XPRM_POP_REF` and `XPRM_POP_REF`).

A.4 Error codes

The module library functions should use the return codes

`XPRM_RT_OK` — to indicate successful execution
`XPRM_RT_ERROR` — to indicate that an error has occurred (interrupts the program run)

APPENDIX B

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count™.

Index

X