SCSK CORPORATION

SCSK

VDMTools

The Dynamic Link Facility

How to contact SCSK:

http://www.vdmtools.jp/	VDM information web site(in Japanese)
http://www.vdmtools.jp/en/	VDM information web site(in English)
http://www.scsk.jp/	SCSK Corporation web site(in Japanese)
http://www.scsk.jp/index_en.html	SCSK Corporation web site(in English)
vdm.sp@scsk.jp	Mail

The Dynamic Link Facility 2.0
— Revised for VDMTools v9.0.2

Contents

1 Introduction

This manual describes a feature of the VDMTools called the Dynamic Link facility. This feature enables that VDM-SL specifications can be combined with code written in C++. This combination enables that a VDM-SL specification during interpretation can use and execute parts that are written in C++. It is called the Dynamic Link facility because the C++ code is dynamically linked together with the Toolbox.

Use of this combination can be of interest in cases where only part of a system is developed using VDM-SL. It can be valuable if:

- a new component should be developed using VDM-SL for a system which already has been implemented;
- a system being specified in VDM-SL needs to make use of features which are not available in VDM-SL itself; and
- it is desirable only to develop a part of the system using VDM-SL.

With this Dynamic Link facility it is possible to investigate the interaction of the parts specified in VDM-SL with parts implemented in C++ at an early stage in the development process.

1.1 Using This Manual

This document is an extension to the *User Manual for the VDM-SL Toolbox* [SCSd]. Before continuing reading this document it is recommended to read the *Dynamic Link Modules* section of *The VDM-SL Language* [SCSb]. Knowledge about C++ [Str91] and *The VDM C++ Library* [SCSa] is also assumed.

You do not need to read this entire manual to get started using the Dynamic Link feature. Start by reading Section 2 to get a detailed description of an example using this feature. All files from this example are also present in the appendices. Section 3 describes the components which are involved when using the Dynamic Link facility one by one. Those who are interested in the way this feature has been implemented internally in the Toolbox should consult [FL96].

Appendix A provides detailed information about system requirements.

2 Getting Started

The combination of formal specifications and code written in C++ only becomes feasible by establishing a common framework combining them. The most basic problem is that the values in these two different worlds have different representations. For the two worlds to communicate, it is necessary to convert values from the specification world to the code world and vice versa. By means of small examples this section illustrates the idea of integrating code with a specification in order to obtain the required functionality.

The examples present the integration of trigonometric functions into a specification using the VDM-SL Toolbox. Trigonometric functions are not included in VDM-SL so in order to develop a system which needs such functions the Dynamic Link facility is used.

2.1 The Basic Idea

The aim of this approach is to be able to analyse the combination of specification and implementation. By combining code and specification the main intention is to provide a prototyping facility by integrating the execution of code into the interpretation of a specification. We enable definitions, made at the code level, to be integrated with a formal specification, such that interpretation of a specification code definitions can be executed. Therefore we distinguish between the specification level, which refers to the VDM-SL specification and integration parts made at within the specification, and the code level, which relates to the code and its integration for the execution.

Figure 1 shows the components involved in combining specification and code. Firstly there are the VDM-SL specification part and the external C++ code part, these are the light grey boxes in the figure. Secondly there is the part that interface the specification and code parts, this is the dark grey boxes in the figure. This interface consists of a part at specification level (called *Dynamic Link modules* or simply *DL modules*) and a part at code level (called *type conversion functions*)

A DL module describes the VDM-SL interface for every definition of the C++ code that will be used in the VDM-SL specification. This information is described as VDM-SL function, operation, and value definitions.

As the C++ code and the VDM-SL Toolbox have different representations for

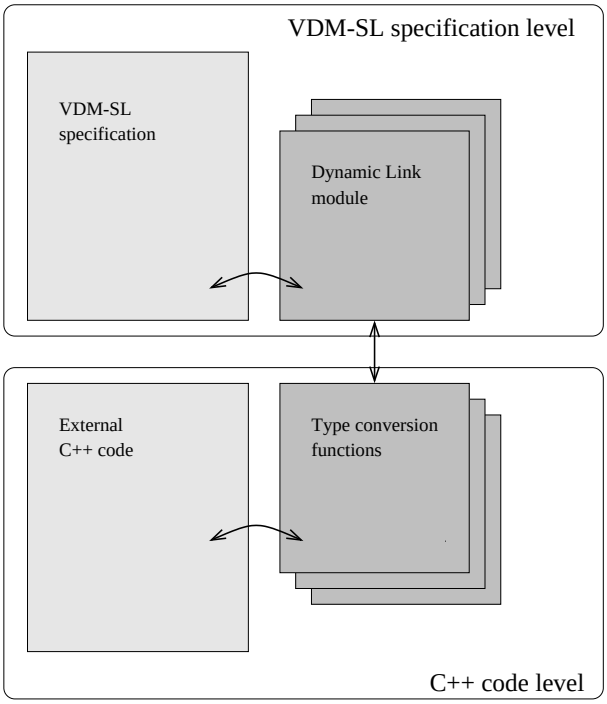

Figure 1: Combination of code and VDM-SL specification

their values, type conversion functions to transform between the tow representations must be defined. These must be defined at the code level. A number of conventions for these type conversion functions have been defined. Generally, the values that are exchanged are of types from the VDM C++ Library. (The VDM C++ library contains classes implementing all the VDM types such as Map, Set, Sequence, Char, Bool, etc. This library is described in detail in [SCSa] and defined in the library file `libvdm.a`, which is included in the Toolbox distribution.)

The convention for calling a C++ function taking parameters and returning a value is as follows:

- The parameters are passed as one value of type "Sequence" from the VDM C++ Library, with each parameter being an element of the sequence: the first argument being the first element, the second argument being the second element etc.
- The returned value is of type Generic (which any VDM Library class type can be converted to).

For calling C++ values and constants a function converting and returning the value/constant must be defined.

Beside that type conversion functions to convert between the VDM C++ values and the C++ code must be defined, no modifications of the C++ code need to be performed.

2.2 An Example with Trigonometric Functions

In order to illustrate the idea an example which integrates trigonometric functions into a specification is presented.

The C++ code, which is to be integrated into a VDM-SL specification, is given by the mathematical standard C library called *math*. The examples make use of implementations of the trigonometric functions *sin*, *cos* and of the constant *pi*. First, a specification which applies these definitions is presented and then the extensions at specification level as well as at code level are shown.

The VDM-SL Specification

The following VDM-SL specification gives an example of how the definitions of the DL module `MATHLIB` are imported into the module `CYLINDER`. Notice how the import from a DL module is identical to the import from an ordinary module, i.e. the module importing does not know whether the imported module is an ordinary module or a DL module.

```
module CYLINDER

imports
  from MATHLIB
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

    values
      ExtPI : real

 definitions
   functions
     CircCyl_Vol : real * real * real -> real
     CircCyl_Vol (r, h, a) ==
       MATHLIB‘ExtPI * r * r * h * MATHLIB‘ExtSin(a)

end CYLINDER
```

The module `CYLINDER` imports the functions `ExtCos` and `ExtSin` and the value `ExtPI` from the DL module `MATHLIB`. The function `CircCyl_Vol` evaluates the volume of a circular cylinder and makes use of the constant `ExtPI` as well as the function `ExtSin`.

The Interface at VDM-SL Level

As mentioned above, the interface between code and specification has to be provided at two different levels. The interface at the VDM-SL level declares the VDM-SL types of the functions that are exported to the outside world. The interface at the code level is based on the definition of the above mentioned type conversion functions.

At the VDM-SL level the interface looks like:

```
dlmodule MATHLIB

  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

    values
      ExtPI : real

    uselib
     "libmath.so"

end MATHLIB
```

A DL module must always contain an export section which declares all constructs that are available to the outside world. The constructs from the export section can be imported by other modules. The export section consists of the signatures for function and operation definitions and the type information for value definitions defining the VDM-SL interface to the C++ code. A value declaration in a DL module relates either to a constant or a variable definition in the code.

The `uselib` field contains the name of the shared object file, which contains the code accessed through the DL module.

The Interface at C++ Level

The interface at the code level is developed in C++ and consists of a number of declarations and the definitions of the type conversion functions. The declaration part includes the standard mathematical library. The type conversion functions convert the VDM C++ sequence value used by the Toolbox to values accepted by the C++ code and vice versa. A generic VDM C++ type can have an underlying value of any VDM type (e.g. the generic VDM C++ type `Sequence` represents a VDM-SL sequence which can contain arbitrary VDM elements.)

The interface looks like:

```
#include "metaiv.h"
```

```
#include <math.h>

# Platform specific directives/includes...

extern "C" {
  Generic ExtCos(Sequence sq);
  Generic ExtSin(Sequence sq);
  Generic ExtPI ();
}

Generic ExtCos(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( cos(rad) ));
}

Generic ExtSin(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( sin(rad) ));
}

Generic ExtPI (Sequence sq)
{
  return(Real(M_PI));
}
```

Notice that *ExtPI* is defined as a VDM-SL value in the VDM-SL interface, but as a function in the code level interface.

The Toolbox puts the arguments of the called function into a value of the generic VDM C++ type Sequence and passes it to the type conversion function. The type conversion function extracts and converts the elements of the sequence into values required by the integrated C++ code. For example, the type conversion function `ExtSin` extracts the first element of the sequence and casts it to the VDM C++ type `Real`, which is casted automatically to the C++ type `double`. This is given as argument to the `sin` function at C++ level and the result is converted into a VDM C++ value and returned to the Toolbox.

In this case a C standard library was used as code, but it could as well have

been a user-defined C++ package. The approach is open to any kind of module developed in C++. The user is only required to develop a DL module and to define type conversion functions. Note that the type conversion functions must be enclosed in an `extern "C"` linkage specification such that C++ parameter type mangling is not made part of the function symbol name that is stored in the shared object library.

Creating A Shared Library

In order to create a dynamically linked library it is required that the type conversion functions are enclosed in an `extern "C"` linkage specification as above. In addition, it is required that the `metaiv.h` header file is included as above. Then the C++ code must be compiled with one of the supported compilers (see section A), using the necessary flags to generate a shared library. It is recommended to record the way the library can be created in a Makefile; example Makefiles showing the requisite flags are shown in Appendix B.3. For Solaris 10 this example would have a Makefile such as[1]:

```
all: libcylio.so libmath.so

%.so:
        ${CXX} -shared -mimpure-text -v -o $@ -Wl,-B,symbolic\
        $^  ${LIB}

libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
        ${CXX} -c -fpic -o $@ $< ${INCL}

tcfcylio.o: tcfcylio.cc
        ${CXX} -c -fpic -o $@ $< ${INCL}

libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
        ${CXX} -c -fpic -o $@  $<  ${INCL}
```

[1]Under the assumption that the interface at the C++ level is placed in a file called `tcfmath.cc`.

The options to be used under other supported platforms/compilers can be seen in Section 3.7.

Running the Example

Having created the shared library `libmath.so` it is now possible to start the Toolbox. The `CYLINDER` module and the `MATHLIB` DL modules can be read into the Toolbox as any other module by syntax checking them. Initialising the specification now will establish the link to the shared library such that it is possible to interpret the `CircCyl_Vol` function, where the calls to `MATHLIB‘ExtPI` and `MATHLIB‘ExtSin` are calls to the library. The VDM-SL parts can also be debugged using the **VDMTools** debugger, whereas the code parts for obvious reasons cannot.

Extending this Example

In the appendices of this document, this small example has been extended with one more DL module. This extra module illustrates how to use a simple I/O interface for the input of the dimensions of a circular cylinder. Other examples from the Examples Repository (http://www.csr.ncl.ac.uk/vdm/examples/examples.html) illustrate the use of a more general user interface using Tcl/Tk.

3 Dynamic Link Components

This section describes the different components which are involved when you are using the Dynamic Link facility, one by one. The intension is to enable you to use this section as a kind of reference guide after having read the previous section once.

In this document we have used a general name convention. We would recommend you to make a convention yourself. It makes it much easier to find out where constructs are defined. The name convention we has used have a “`Ext`” prefix for all constructs which are defined externally (i.e. in the C++ code). All files with conversion functions have a “`tcf`” prefix. Finally, on Unix all dynamically linked libraries have a “`lib`” prefix and they all have an `.so` extension; on Windows all dynamically linked libraries have a `.dll` extension.

3.1 Dynamic Link Modules

A *DL module* provides the VDM-SL interface to the code parts. Beside the module name it contains an import section, an export section, and the name of the library (containing the C++ code) to link dynamically with.

- The import section, which is optional, describes VDM-SL types used within the DL module. An example of this is provided in the `CYLIO` DL module of Appendix B.1.2.
- The export section describes the functions, operation and values available from the DL module. Notice that unlike ordinary modules in DL modules all exported constructs must be explicitely described, i.e. no `exports all` exists.
- The name of the shared library describes which library to link dynamically with. This can be left our, in which case the specification can by syntax and type checked, but not initialised or interpreted.

  Section 3.5 describes how to define where the Toolbox should search for the library.

The precise syntax and semantics of the DL modules is described in more detail in the *Dynamic Link Modules* section of *The VDM-SL Language* [SCSb].

The difference between functions and operations is the same for DL modules as for ordinary modules: functions must return a value and must not have any side effects. In contrast, operations may have side effects, and need not return any values. Naturally, this is simply a pragmatic difference which the Toolbox cannot check.

3.2 Type Conversion Functions

The purpose of the type conversion functions is to convert the values used by the Toolbox (these are objects of the *VDM C++ Library*) into values required by the integrated code and vice versa. A type conversion function must always take as argument an object of the *VDM C++ Class* type `Sequence` in order to handle functions with different arity. It returns either an object of the *VDM C++ Class* `Generic` or `void`. A type conversion function must be defined for each construct that is exported from a Dynamic Link module.

If you wish to integrate a function which is defined in C++ code, (e.g. `double Volume (double radius, double height)`), a type conversion function must be defined. In this case, the DL module should contain a corresponding function or operation definition, say `functions ExtVolume: real * real -> real`. The Toolbox puts the evaluated arguments, given by the interpreted specification, into an object of the class `Sequence`, and passes it to the type conversion functions. The definition of this small example is:

```
Generic ExtVolume (Sequence sq1)
{
  double rad, height;
  rad = (Real) sq[1];
  height = (Real) sq[2];
  return( (Real) Volume(rad, height) );
}
```

The arguments for the function `Volume` are extracted from the sequence and converted into double values. The return value from `Volume` is converted into an object of the class `Real` which is automatically casted into a `Generic`.

Table 1 provides an overview of related definitions in DL modules, type conversion functions, and C++ code, i.e. how values are represented at the different levels. Appendix B contains the type conversion functions for the trigonometric example.

3.3 Converting Records

Special attention needs to be given to records. In the VDM-SL specification records are tagged with a name, whereas in the VDM C++ Library they are tagged with an integer. These tags are refered to as "symbolic names" and "numeric tags" respectively in the remaining of this section.

The VDM C++ Library can maintain a map (called the Record Info Map), mapping numeric tags to symbolic names. Any symbolic name which is sent to or returned from the code part of a dynamically linked module must have been registered in this map.

This registering must be defined in a function named `InitDLModule`. If this function exists the Toolbox will call it during initialisation (when executing the `init` command). The example below illustrates how the Record Info Map is built within the InitDLModule function:

DL module definition	*type conversion function*	*C++ code*
value definition	function with no argument and result type `Generic`	variable or constant

`values`
`  ExtLength: real`
`  ExtMax:nat`
	`Generic ExtLength ()`
	`Generic ExtMax ()`
		`double Length = 5.0;`
		`const int max = 20;`

function or operation definition	function with argument `Sequence` and result `Generic`	function with result different from `void`

`functions`
`  ExtVolume: real * real -> real`
or
`operations`
`  ExtVolume: real * real ==> real`
	`Generic ExtVolume(Sequence sq1)`	`double Volume(double radius,`
		`  double height)`

operation without result	function with argument `Sequence` and result type `void`	function with result `void`
`operations` `  ExtShowItem: nat ==> ()`	`void ExtShowItem(Sequence sq1)`	`void ShowItem (int item)`

Table 1: Value representations at different levels.

- For each symbolic name used, a corresponding numeric tag must be added in the Record Info Map. This is performed by executing:

  ```
  VDMGetDefaultRecInfoMap().NewTag(numtag, recordsize);
  ```

  where `numtag` is the numeric tag, and `recordsize` is the number of fields in the record.

- The symbolic name must be bound to the numeric tag. This is performed by executing:

  ```
  VDMGetDefaultRecInfoMap().SetSymTag(numtag, "symbname");
  ```

  where `numtag` still is the numeric tag, and `symbname` is the symbolic name. Noce that the symbolic name must be encapsulated in quotes.

In the example the numeric tag and record size are defined as constants (`TagA_X` and `TagA_X_Size`).

The examples:

```
extern "C" void InitDLModule(bool init);
const int TagA_X = 1;
const int TagA_X_Size = 2;
void InitDLModule(bool init) {
  if (init) {
    VDMGetDefaultRecInfoMap().NewTag(TagA_X, TagA_X_Size);
    VDMGetDefaultRecInfoMap().SetSymTag(TagA_X, "A`X");
  }
}
```

When the Toolbox is creating a Record value with name `A‘X` to be sent to the C++ code it will find the numeric tag corresponding to the sumbolic name `A‘X` in the Record Info Map of the DL Module.

Notice the following issues when creating a Record Info Map:

- The symbolic name of the `SetSymTag` command must be an existing record name qualified with the corresponding module name, and the record size of the `NewTag` command must be the exact number of fields of the record.
- Each symbolic tag must exists only once in the Record Info Map.

3.3.1 Using Records in combination with the Code Generator

The Code Generator creates code that sets up the Record Info Map for code generated modules. Thus if generated code is compiling and linking into a dynamically linked library, the InitDLModule function can simply call the function `init_Module()`, to define the records in *Module* in the Record Info Map.

3.4 Converting Tokens

The token type is exported from the toolbox as a C++ class `Record`. However, the tag of token records is always equal to `TOKEN`, which is a macro declared in the file `cg_aux.h`, and the number of fields in token records is always equal to 1. If you want to send a token from the external code to the toolbox, the VDM-SL value `mk_token(<HELLO>)` can e.g. be constructed in the following way:

```
Record token(TOKEN, 1);
token.SetField(1, Quote("HELLO"));
```

3.5 The `uselib` Path Environment

The name of the library is given with the `uselib` option in the specification. This location can be given in several ways:

- A full path name for the library (e.g. /home/foo/libs/libmath.so)
- Without path, but with the environment variable `VDM_DYNLIB` set. The library is searched for in every directory name in the `VDM_DYNLIB` environment variable. The environment variable could look like this: `/home/foo/libs:/usr/lib:.` (Note that if you set the variable, you need a '.' if you want the search to look in the current directory too.)
- Without path, and without the `VDM_DYNLIB` environment variable set. This means that the library is supposed to be located in the current directory.

3.6 DL Module Initialisation

The first time the Toolbox command "init" is used all modules will be loaded. The second time, the currently loaded modules are first unloaded, then the modules

are loaded again.

3.6.1 Module Loading

When a shared library file is loaded by the Toolbox, the following will happen:

- The global variables will be initialised.
- The toolbox executes the function InitDLModule(true) in each loaded module.

3.6.2 Module Unloading

When a shared library file is unloaded by the Toolbox, the following will happen:

- The toolbox executes the function InitDLModule(false) in each loaded module.
- The global variables will be destructed.

3.7 Creating a Shared Library

The shared library must be compiled using the proper compiler as required in Section A. In order to create an executable shared library the code with the type conversion functions must fulfill the following requirements:

- The type conversion functions must be enclosed in an `extern "C"` linkage specification, otherwise the functions will not be reachable from outside the library.
- The `metaiv.h` header file, which is part of the `VDM C++ Library`, must be included in the file with the type conversion functions as it contains prototypes of the type specific functions of the *VDM C++ Library*.

See the example makefiles in B.3 for the actual compiler flags to use.

On Unix a shared library should have the file extension `".so"` and a prefix `"lib"`. (A version number for the shared library is not required.) On Windows a shared library should have the file extension `".dll"`

References

[FL96] Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL Specifications with C++ Code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179–194. Springer-Verlag, March 1996.

[SCSa] SCSK. *The VDM C++ Library*. SCSK.

[SCSb] SCSK. *The VDM-SL Language*. SCSK.

[SCSc] SCSK. *The VDM-SL to C++ Code Generator*. SCSK.

[SCSd] SCSK. *VDM-SL Toolbox User Manual*. SCSK.

[Str91] B. Stroustrup. *The C++ Programming Language, 2nd edition*. Addison Wesley Publishing Company, 1991.

A System Requirements

To use the Dynamic Link feature the *VDM-SL Toolbox* including a license to the Dynamic Link feature (a line with the `vdmdl` feature must be present in the license file) and the *VDM C++ Library* are required. Furthermore a compiler is required in order to create an executable shared library of the integrated code.

This feature runs on the following combinatins:

- Microsoft Windows 2000/XP/Vista and Microsoft Visual C++ 2005 SP1
- Mac OS X 10.4, 10.5
- Linux Kernel 2.4, 2.6 and GNU gcc 3, 4
- Solaris 10

The compiler is required in order to create an executable shared library of the integrated code.

For installation of the Toolbox itself see Section 2 in [SCSd] and for installing the *VDM C++ Library* see Section 2 in [SCSc].

B Overview of the Trignometric Example

In this example a simple user interface to trigonometric functions in a specification is integrated in order to calculate the volume of a circular cylinder. This example is available with the distribution in the directory `example`. To run the example do the following:

- Edit the appropriate makefile, and set the paths at the top to where your libraries are.
- Compile the libraries

  Linux make -f Makefile.Linux

  Solaris 10 make -f Makefile.solaris2.6

  Microsoft Windows 2000/XP/Vista make -f Makefile.win32

- start the Toolbox, read the specifications, initialise and call the function. In the command line version of the Toolbox this would look as:

```
vdm> r cylio.vdm
Parsing "cylio.vdm" ... done
vdm> r cylinder.vdm
Parsing "cylinder.vdm" ... done
vdm> r math.vdm
Parsing "math.vdm" ... done
vdm> init
Initializing specification ...
vdm> p CYLINDER`CircCylOp()


 Input of Circular Cylinder Dimensions
  radius: 10
  height: 10
  slope [rad]: 1


 Volume of Circular Cylinder

Dimensions:
   radius: 10
   height: 10
   slope: 1

 volume: 2643.56

(no return value)
```

The *VDM-SL* document consists of a module *CYLINDER* which imports the DL modules *MATHLIB* and *CYLIO*. The DL module *MATHLIB* provides an interface to the trigonemetric function *sinus* and the value π. The module *CYLIO* contains a function for setting the dimension of the circular cylinder and for printing its volume. The module *CYLINDER* also defines a record for representing dimensions of a circular cylinder which is imported by the DL modules.

For each DL modules there exists a corresponding shared library: *MATHLIB* to `libmath.so/math.dll` and *CYLIO* to `libcylio.so/cylio.dll`.

The shared library `libmath.so/math.dll` provides the trigonometric functions *sine*, *cosine* and the value π which all are defined in the C standard library. The code consists of the type conversion functions. The shared library `libcylio.so` (or `cylio.dll` on Windows) provides simple user interface functions: `GetCircCyl` asks for the dimensions of the cylinder and `ShowCircCylVol` prints the cylinder dimensions and its volume on the screen. The cylinder dimensions are represented by a structure `CircCyl` which has an equivalent definition in the specification, the record type *CircCyl*.

B.1 The Specifications

This appendix contains the module specifications.

B.1.1 The Module *CYLINDER*

```
module CYLINDER
  imports
    from MATH
      functions
        ExtCos : real -> real;
        ExtSin : real -> real
      values
        ExtPI : real,

    from CYLIO
      functions
        ExtGetCylinder : () -> CircCyl

      operations
        ExtShowCircCylVol : CircCyl * real ==> ()

   exports
     operations
       CircCylOp:() ==> ()
     types
       CircCyl

  definitions
    types
```

```
      CircCyl :: rad : real
                 height : real
                 slope : real

    functions
      CircCylVol : CircCyl -> real
      CircCylVol(cyl) == MATH`ExtPI * cyl.rad * cyl.rad *
                         cyl.height * MATH`ExtSin(cyl.slope);

    operations
      CircCylOp : () ==> ()
      CircCylOp() == ( let cyl = CYLIO`ExtGetCylinder() in
                         let vol = CircCylVol(cyl) in
                           CYLIO`ExtShowCircCylVol(cyl, vol))


end CYLINDER
```

B.1.2 The Dynamic Link Modules *MATHLIB* and *CYLIO*

```
dlmodule MATH
  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

   values
      ExtPI : real

   uselib
      "math.dll"

end MATH

dlmodule CYLIO
  imports
```

```
    from CYLINDER
      types
        CircCyl

   exports
     functions
       ExtGetCylinder : () -> CYLINDER`CircCyl

     operations
       ExtShowCircCylVol : CYLINDER`CircCyl * real ==> ()

   uselib
      "cylio.dll"

end CYLIO
```

B.2 The Shared Libraries

This appendix contains the source files used to build the shared libraries.

B.2.1 The MATHLIB shared library

```
//------------------------------------------------------------------
// tcfmath.cc   type conversion functions for libmath.so
//------------------------------------------------------------------
#include "metaiv.h"
#include <math.h>

#ifdef WIN32
#define DLLFUN __declspec(dllexport)
#define M_PI   3.14
#else
#define DLLFUN
#endif
```

```
extern "C" {
  DLLFUN void InitDLModule(bool init);
  DLLFUN Generic ExtCos(Sequence sq);
  DLLFUN Generic ExtSin(Sequence sq);
  DLLFUN Generic ExtPI ();
}

void InitDLModule(bool init)
{
  // This function is called by the Toolbox when modules are
  // initialised and before they are unloaded.
  // init is true on after load and false before unload.
}

Generic ExtCos(Sequence sq) {
  return (Real( cos((Real)sq[1]) ));
}

Generic ExtSin(Sequence sq) {
  return (Real( sin((Real) sq[1]) ));
}

Generic ExtPI () {
  return(Real(M_PI));
}
```

The type conversion function `ExtSin` takes as argument an object of the class `Sequence`, extracts the argument and converts it into an object of the class `Real`. This object is casted automatically in a `double` value by application of the *sin* function. The result value is converted into a Meta-IV value and returned to the

Toolbox.

B.2.2 The CYLIO shared library

This section contains the sources of the shared library `libcylio.so/cylio.dll`. The file `cylio.cc` provides a simple interface for the input of the dimensions of the circular cylinder and the printout of the volume.

The file `tcfcylio.cc` contains the type conversion functions. The type conversion functions also show the transformation of the circular cylinder values between the different representations of the type "CircCyl" used in C++ and in the Toolbox.

```
//-----------------------------------------------------------------
// cylio.h        Definition of the structure used.
//-----------------------------------------------------------------
struct CircularCylinder {
  float rad;
  float height;
  float slope;
};

typedef struct CircularCylinder CircCyl;

extern CircCyl GetCircCyl();
extern void ShowCircCylVol(CircCyl cylinder, float volume);


//-----------------------------------------------------------------
// cylio.cc        Circular Cylinder simple I/O functions
//-----------------------------------------------------------------
#include <iostream.h>
#include "cylio.h"

CircCyl GetCircCyl() {
  CircCyl in;

  cout <<"\n\n Input of Circular Cylinder Dimensions";
  cout <<"\n  radius: ";
  cin >> in.rad;

  cout <<"  height: ";
```

```
  cin >> in.height;

  cout <<"  slope [rad]: ";
  cin >> in.slope;

  return in;
}


void ShowCircCylVol(CircCyl cyl, float volume) {
  cout << "\n\n Volume of Circular Cylinder\n\n";
  cout << "Dimensions: \n   radius: " << cyl.rad;
  cout << "\n   height: " << cyl.height;
  cout << "\n   slope: "<< cyl.slope << "\n\n";
  cout << " volume: " << volume<< "\n\n";
}
```

```
//-----------------------------------------------------------------
// tcfcylio.cc  type conversion functions for circular cylinder io
//-----------------------------------------------------------------

#include "metaiv.h"
#include "cylio.h"

#ifdef WIN32
#define DLLFUN __declspec(dllexport)
#else
#define DLLFUN
#endif

extern "C" {
  DLLFUN void InitDLModule(bool init);
  DLLFUN Generic ExtGetCylinder(Sequence sq1);
  DLLFUN void ExtShowCircCylVol(Sequence sq1);
```

```
}

const int tag_CYLINDER_CircCyl = 1;
const int size_CYLINDER_CircCyl = 3;

void InitDLModule(bool init)
{
  if (init) {
    VDMGetDefaultRecInfoMap().NewTag(tag_CYLINDER_CircCyl,
                                     size_CYLINDER_CircCyl);
    VDMGetDefaultRecInfoMap().SetSymTag(tag_CYLINDER_CircCyl,
                                        "CYLINDER`CircCyl");
  }
}

Generic ExtGetCylinder(Sequence sq1)
{
  CircCyl cyl;
  Record Rc(tag_CYLINDER_CircCyl,size_CYLINDER_CircCyl );

  cyl = GetCircCyl();   // input of cylinder dimension
  Rc.SetField(1, (Real)cyl.rad); // conversion in VDM C++ record class
  Rc.SetField(2, (Real)cyl.height);
  Rc.SetField(3, (Real)cyl.slope);
  return(Rc);    // return Record to the interpreter process
}

void ExtShowCircCylVol(Sequence sq1)
{
  CircCyl cyl;
  Record Rc(tag_CYLINDER_CircCyl,size_CYLINDER_CircCyl);
  float vol;

  // extract cylinder dimension and volume from sequence
  Rc =  sq1[1];
  vol = (Real) sq1[2];

  // convert Record in a C++ structure
  cyl.rad = (Real) Rc.GetField(1);
  cyl.height = (Real) Rc.GetField(2);
  cyl.slope = (Real) Rc.GetField(3);
```

```
  ShowCircCylVol(cyl, vol); //make output
  return;
}
```

B.3 Makefiles

B.3.1 Linux

Here is an example makefile for Linux.

```
##---------------------------------------------------------------
##                    Make file for Linux
##---------------------------------------------------------------

#VDMLIB  = /opt/toolbox/cg/lib
VDMLIB  = /local2/paulm/vice
#INCL    = -I/opt/toolbox/cg/include
INCL    = -I/local2/paulm/vice

CC      = /opt/gcc-3.0.1/bin/g++
LIB     =  -L$(VDMLIB) -lvdm -lm # -lstdc++ -lgcc
# -Wl,-Bstatic

## Nothing below this line should be changed.

all: libcylio.so libmath.so

%.so:
${CC} -shared -fpic $(EXCEPTION) -v -o $@ $^ $(LIB)

libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
${CC} -c $(EXCEPTION) -fpic -o $@ $< ${INCL}

tcfcylio.o: tcfcylio.cc
${CC} ${INCL} -c $(EXCEPTION) -fpic -o $@ $<
```

```
libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
${CC} -c $(EXCEPTION) -fpic -o $@  $<  ${INCL}


clean:
rm *.o *.so

# target for debuging
debug: cylio.o tcfcylio.o main.cc
${CC} -c main.cc -o main.o ${INCL}
${CC} -o debug cylio.o tcfcylio.o main.o $(LIB)
```

B.3.2 Solaris 2.6

Here is the example makefile for Solaris 10

```
##-------------------------------------------------------------
##                    Make file for Solaris 2
##-------------------------------------------------------------
VDMLIB  = /opt/toolbox/cg/lib
INCL    = -I/opt/toolbox/cg/include

## Compiler flags
CC      = g++
LIB     = -L$(VDMLIB)  -lvdm -lm

all: libcylio.so libmath.so

%.so:
${CC} -shared -mimpure-text -v -o $@ -Wl,-B,symbolic $^  ${LIB}

libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
${CC} -c -fpic -o $@ $< ${INCL}

tcfcylio.o: tcfcylio.cc
```

```
${CC} -c -fpic -o $@ $< ${INCL}

libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
${CC} -c -fpic -o $@  $<  ${INCL}


clean:
rm *.o *.so
```

B.3.3 Windows

Here is the example makefile for Microsoft Windows, using GNU make.

```
##-------------------------------------------------------------
##                    Make file for Windows 32bit
##             This Makefile can only be used with GNU make
##-------------------------------------------------------------

CC       = cl.exe
CFLAGS   = /nologo /c /MT /W0 /GD /GX /D "WIN32" /D "_USRDLL" /TP
INCPATH  = /I//hermes/georg/toolbox/winnt

LINK     = link.exe
LPATH    = /LIBPATH:C:/work
LFLAGS   = /dll /incremental:no /DEFAULTLIB:vdm.lib

# IMPLICITE RULES

%.obj: %.cc
$(CC) $(CFLAGS) $(INCPATH) /Fo"$@" $<

%.dll:
$(LINK) $(LPATH) $(LFLAGS) /out:"$@" $^

# TARGETS

all: math.dll cylio.dll
```

```
math.dll: tcfmath.obj

cylio.dll: tcfcylio.obj cylio.obj

clean:
rm -f math.lib math.exp math.dll
rm -f cylio.lib cylio.exp cylio.dll
rm -f tcfmath.obj tcfcylio.obj cylio.obj
rm -f *~
```

C Trouble Shooting

Once you have started creating your own type conversion functions, you will discover that suddenly strange error messages from the VDM C++ library may occur. This appendix will help you providing an idea of how to debug your libraries.

The best thing is to start `gdb` as usually, and debug through the code, to see where it goes wrong. This cannot be done straight away, since the symbols defined in the libraries are not known, when `gdb` is started.

The solution is first to debug the specification with `vdmde`, and when it has been discovered which external function goes wrong, a `main()` which calls that particular function can be created and linked together with your libraries. The main procedure could look like this:

```
#include "metaiv.h"

extern "C" {
  Generic ExtGetCylinder(Sequence);
}

main() {
```

```
  Sequence sq; // empty sequence to call ExtGetCylinder with.
  Generic result;

  result = ExtGetCylinder(sq);
}
```

C.1 Segmentation Fault

Another problem you may have is that the toolbox crashes with a segmentation fault. This might very well be a problem in your external library. Again, `vdmde` is started in order to see in which function the problems occur.

There are three obvious things which can go wrong in the interface between the Toolbox and your code resulting in such an error:

1. Your functions do not have the correct signature:
   `void` *function*`(Sequence` *args*`)`
   or `Generic` *function*`(Sequence` *args*`)`
2. A function which should return `Generic` does not return any value.
3. The external code contain global uninitialised values or objects (see Section C.3).

C.2 Using Tcl/Tk

If you wish to use Tcl/Tk in a dynamically linked library together with the Toolbox you should look at the other examples which have been made for this combination [2]. It is important that you do not call `Tk_MainLoop` if you are using `vdmgde` (the graphical version of the Toolbox) because `vdmgde` itself is implemented with Tcl/Tk and therefore it will enter a deadlock situation where Tcl/Tk will be waiting for the main Toolbox window to close in case `Tk_MainLoop` is called.

One solution (which is used in the other examples) is to create a loop which calls `Tk_DoOneEvent` and terminates when the Tcl code has set a Tcl variable `Done`

[2] There is no exercise being offered now.

non zero. Initially, the Tcl code sets `Done` to zero and when the dialog window is closed `Done` is set to 1.

C.3 Global Objects and initialisation of Global Values

On all supported platform (Windows, Mac, Linux, Solaris) C++ object constructors are called when the shared object is compiled and linked as described above.

C.4 Standard input and output under Microsoft Windows

Since the graphical version of the Toolbox runs as a GUI application under Windows, standard input and output streams are not available. Therefore shared libraries that attempt to use `cin`, `cout` and `cerr` will not work as expected.