SCSK CORPORATION

SCSK

VDMTools

The Dynamic Link Facility for VDM++

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 for VDM++ 2.0
— Revised for VDMTools v9.0.2

© COPYRIGHT 2013 by SCSK CORPORATION

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice.

Contents

1 Introduction

This document is an extension to the *User Manual for the VDM++ Toolbox* [SCSd]. Knowledge about C++ [Str91] and The *VDM C++ Library* [SCSa] is also assumed. Throughout the document we use the term "DL Classes" ("dynamic link classes") to mean VDM++ classes for which external code is to be executed.

You do not need to read this entire manual to get started using the Dynamic Link feature. Start by reading Section 2 to get an overall description of this feature. Section 3 shows an example of using the feature; all files from this example are also present in the appendices. Section 4 demonstrates how code is generated from DL Classes by the C++ Code Generator. Section 5 describes the individual components which are involved when using the Dynamic Link facility.

Appendix A provides detailed information about system requirements.

2 Overview

In this section an overview of the use of DL classes is presented. The idea here is to give an informal idea of how to use the facility. More detailed information is presented in the Reference Guide in Section 5.

A diagram showing the approach is given in Figure 1. As can be seen from this diagram, a number of different components need to be provided to enable the facility:

- For each C++ class which is to be accessed at the VDM++ level, a DL Class must be written at the VDM++ level.
- An interface layer needs to be written in C++. This translates method calls at the VDM++ level into calls to the external code and translates the results into *VDM C++ Library* types. This should be compiled as a shared library (on Windows, a `.dll`).

Each of these components is described in more detail below.

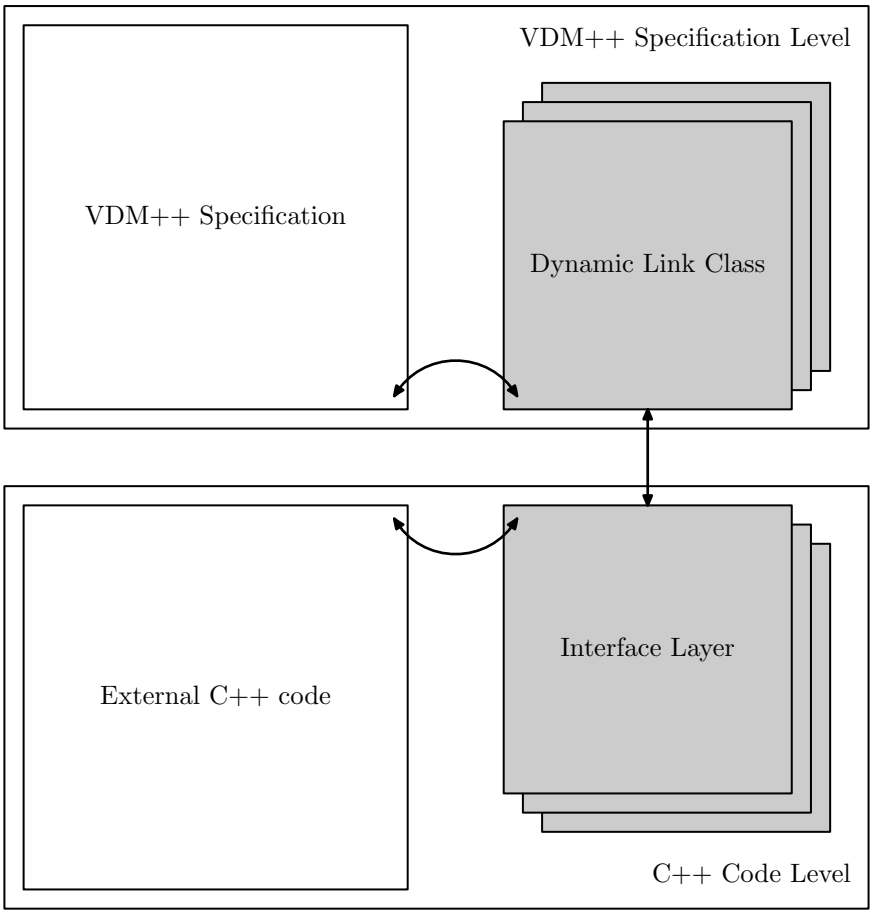

Figure 1: Combination of code and VDM++ specification

2.1 Specifying a DL Class in VDM++

To specify a class as being a DL class in VDM++, the keyword dlclass should be used instead of class. The DL Class must contain a directive indicating the location of the shared library implementing the DL class. This is specified using the uselib directive. Thus a simple DL class might be

```
dlclass EmptyDLClass

uselib "EmptyDLClass.so"[1]

end EmptyDLClass
```

Of course, this is not a very interesting class since it contains no functionality! To expose methods in the external code, methods with body is not yet specified should be defined at the VDM++ level. For example:

```
dlclass SimpleDLClass

uselib "SimpleDLClass.so"

operations

public CallExternal : nat ==> nat
CallExternal (n) == is not yet specified;

end SimpleDLClass
```

Calls to `SimpleDLClass‘CallExternal` will then, via the shared library, be directed to the external code corresponding to `SimpleDLClass`.

2.2 Interfacing with a DL Class

The user must provide an interface between the VDM++ DL Class and the external code to which the DL Class corresponds. This interface is referred to as the interface layer and falls into three parts:

[1] Note that shared libraries have file type `.so` on Unix whereas on Windows they are of type `.dll`

1. Functions for creating and deleting external objects must be provided. These are called respectively `DlClass_new` and `DlClass_delete`;
2. A function for redirecting method calls must be defined, which is called `DlClass_call`;
3. Each external class must be wrapped as a subclass of the class `DlClass`. That is, every external class must implement the `DlClass` interface.

The prototypes for these functions and classes are defined in the file `dlclass.h` which is supplied with the Toolbox distribution. They can also be found in Appendix B.

2.3 Creating a Shared Library

Having created the interface layer, a shared library must be constructed. This is done by compiling the interface layer and the external code into a single shared library. On Unix using GNU g++, this is achieved using the flags `-shared -fpic`. On Windows using Microsoft Visual C++ it is achieved using the link flags `/dll /incremental:no`. In both cases the VDM C++ library must be linked in since the interface layer must use the types defined in this library. See the example Makefiles that are provided in Appendices D and E.

3 Example

In this section we present portions of a small example of how to write a DL Class and interface layer code. The example is based on an external library – "MAPM" – for arbitrary precision mathematical calculations. This library is freely available for download from `http://www.tc.umn.edu/~ringx004/mapm-main.html`.

Since the VDM++ interpreter uses finite precision arithmetic, it is important that infinite precision mathematical objects are never handled directly by the Toolbox since by doing so the precision of the mathematical values will be compromised.

3.1 The VDM++ Model

We define a class `BigInt` to represent arbitrary precision integers. This defines operations to create `BigInt`s, to convert a texttttBigInt to a string representation,

and to perform addition on `BigInt`s. (Note that the `SetVal` operation is only used to create an initial value.) The class is connected to an interface layer called `bigint_dl.so`.

```
dlclass BigInt

uselib "bigint_dl.so"

operations

public Create : nat ==> BigInt
Create(n) ==
( SetVal(n);
  return self
);

public toString : () ==> seq of char
toString() ==
  is not yet specified;

public plus : BigInt ==> BigInt
plus(i) ==
  is not yet specified;

protected SetVal : nat ==> ()
SetVal(n) ==
  is not yet specified;

end BigInt
```

We can then use `BigInt` in other classes in our specification. For example, the class `BankAccount` uses `BigInt` as the type of one of its instance variables. As shown by the `Init` and `Deposit` operations, manipulation of this object is performed via operations rather than directly.

```
class BankAccount

instance variables
  name : seq of char;
  number : nat;
```

```
  balance : BigInt

operations

public Init : seq of char * nat ==> ()
Init(nname, nnum) ==
( name := nname;
  number := nnum;
  balance := new BigInt().Create(nnum);
);

public Deposit : BigInt ==> ()
Deposit(bi) ==
  balance := balance.plus(bi);

public GetBalance : () ==> BigInt
GetBalance() ==
  return balance

end BankAccount
```

3.2 Interface Layer

The interface layer maps calls at the VDM++ level to C++ function calls. The complete files are included in Appendix C. Here we describe selected excerpts.

For each VDM++ `BigInt` object, there is a corresponding C++ `BigIntDL` object. This is linked to the external MAPM library via a member variable:

```
class BigIntDL : public DlClass
{

  MAPM val;

 public:
 ...
};
```

To see how communication between the VDM++ level and the C++ level is mediated, consider the class function `toString` which returns a string representation

of a BigInt.

```
Sequence BigIntDL::toString()
{
#ifdef DEBUG
  cout << "BigIntDL::toString" << endl;
#endif //DEBUG
  char res[100];
  val.toIntegerString(res);
  return Sequence(string(res));
}
```

This function uses the library's own `toIntegerString` function. Note that since the value is to be returned to the VDM++ Toolbox, a *VDM C++ Library* value must be returned by the function.

More interesting is the case where VDM++ objects are passed as parameters and/or returned as result. In this case, the *VDM C++ Library* class `DLObject` should be used. For example, consider the operation of addition:

```
DLObject BigIntDL::plus (Sequence &p)
{
#ifdef DEBUG
  cout << "BigIntDL::plus" << endl;
#endif //DEBUG
  DLObject result("BigInt", new BigIntDL);
  DLObject arg (p.Hd());
  BigIntDL *resPtr = (BigIntDL*) result.GetPtr(),
           *argPtr = (BigIntDL*) arg.GetPtr();
  resPtr->setVal( val + argPtr->getVal());
  return result;
}
```

Here, a new `DLObject` is created, as an instance of `BigInt`, to store the result of the operation. The argument object is extracted from the parameter list the pointers to the `BigIntDL` objects are extracted from the `DLObjects`, and the result is computed. Finally the result object is returned.

Objects created and returned in this way are subject to the VDM++ Toolbox's normal object maintenance. Thus they will persist until no references exist to them at the VDM++ level, at which point `DlClass_delete` will be called on such objects.

4 Using DL Classes in combination with the C++ Code Generator

DL Classes are treated by the code generator in such a manner as to allow virtually seamless use of generated code with the interface layer and external code. Of course, user implementations for VDM++ operations specified as is not yet specified can be provided in the usual way, thus allowing direct communication with the external code rather than via the interface layer. In this section we illustrate both of these approaches.

4.1 Using Generated Code

The generated code uses the same calling structure as used by DL Classes. To allow this, each class has a public member which is a pointer to a DL Class:

```
class vdm_BigInt : public virtual CGBase {
...
public:
  DlClass * BigInt_dlClassPtr;
}
```

C++ function calls corresponding to operation or function calls to DL Classes are then code generated using this pointer. For example, consider the operation `toString` from Section 3. This would be code generated as a call to `DlClass_call` using `BigInt_dlClassPtr`.

```
#ifndef DEF_BigInt_toString

type_cL vdm_BigInt::vdm_toString () {  Sequence parmSeq_2;
  int success_3;

  return DlClass_call(BigInt_dlClassPtr, "toString", parmSeq_2, success_3);

}
#endif
```

Thus calls within the generated code to `vdm_BigInt::vdm_toString` will be seamlessly handled by the interface layer.

For VDM++ functions or operations that pass or receive objects, the situation is slightly more complicated. This is because the VDM++ Code Generator uses `ObjectRef` from the *VDM C++ Library* to represent objects passed between C++ functions. Therefore the code in the interface layer needs to be able to deal with this. For example, we might modify `BigIntDL::plus` as follows:

```
#ifdef CG
ObjectRef BigIntDL::plus (const Sequence &p)
#else
DLObject BigIntDL::plus (const Sequence &p)
#endif //CG
{
#ifdef DEBUG
  cout << "BigIntDL::plus" << endl;
#endif //DEBUG
  // Extract arguments
  BigIntDL *argPtr = GetDLPtr(p.Hd());

  // Set up result object
#ifdef CG
  ObjectRef result (new vdm_BigInt);
#else
  DLObject result("BigInt", new BigIntDL);
#endif

  BigIntDL *resPtr = GetDLPtr(result);

  // Perform manipulation on pointers, as needed for function
  resPtr->setVal( val + argPtr->getVal());

  return result;
}
```

This definition exploits the function `BigIntDL::GetDLPtr`:

```
#ifdef CG
BigIntDL *BigIntDL::GetDLPtr(const ObjectRef& obj)
#else
BigIntDL *BigIntDL::GetDLPtr(const DLObject& obj)
#endif //CG
```

```
{
#ifdef CG
  vdm_BigInt *objRefPtr = ObjGet_vdm_BigInt(obj);
  BigIntDL *objPtr = (BigIntDL*) objRefPtr->BigInt_dlClassPtr;
#else
  BigIntDL *objPtr = (BigIntDL*) obj.GetPtr();
#endif
  return objPtr;
}
```

4.2 Supplying a User Implementation

Rather than communicate with the external code via the interface layer, the generated code can be made to call the external code directly. This is achieved using the standard mechanism for substituting code in generated code, as described in [SCSc].

For instance, suppose we wish to call the addition operation directly. For this, we need to supply an implementation of `vdm_BigInt::vdm_plus`. In the file `BigInt_userimpl.cc` the function would be defined:

```
#include "bigint_dl.h"

type_ref_BigInt vdm_BigInt::vdm_plus (const type_ref_BigInt &vdm_i)
{
  type_ref_BigInt result (new vdm_BigInt);
  vdm_BigInt *vdm_r_ptr = ObjGet_vdm_BigInt(result),
             *vdm_i_ptr = ObjGet_vdm_BigInt(vdm_i);
  BigIntDL *argPtr = (BigIntDL*) vdm_i_ptr->BigInt_dlClassPtr,
           *thisPtr = (BigIntDL*) BigInt_dlClassPtr,
           *resPtr = (BigIntDL*) vdm_r_ptr->BigInt_dlClassPtr;
  resPtr->setVal(thisPtr->getVal() + argPtr->getVal());
  return result;
}
```

To ensure that the correct version of the function is compiled, the file `BigInt_userdef.h` needs to have the following contents:

```
#define DEF_BigInt_USERIMPL
```

```
#define DEF_BigInt_plus
```

When the files are compiled, the execution uses the user supplied function instead of being directed to the interface layer.

5 Reference Guide

5.1 DL Class Definitions

A DL Class is specified using the syntax

dlclass = 'dlclass', identifier, [inheritance clause]
uselib clause,
[class body],
'end' identifier ;

uselib clause = 'uselib', text literal ;

Note that the syntactic classes not defined here are defined in [SCSb]. A DL Class is treated by the Toolbox just as any other class, with the exception that any call to a function or operation with body is not yet specified is redirected to the external code specified in the uselib path.

5.2 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 to a list of directories in which to search for the library. For example, the environment variable could look like this: `/home/foo/libs:/usr/lib:.` (Note that if you set this variable you need a '.' in the list as in this example if you want the search to include the current directory.)
- Without path and without the `VDM_DYNLIB` environment variable set. This means that the library is supposed to be located in the current directory.

5.3 The Function `DlClass_call`

This function is used to redirect method calls at the VDM++ level to appropriate method calls at the external code level. It takes as parameters:

`DlClass* c` A pointer to the object on which the call is being made;

`const char* name` The name of the method being called;

`const Sequence& params` The parameters to the method call, passed as *VDM C++ Library* values;

`int& success` A reference parameter to a success flag. Setting this flag to `0` indicates failure; setting it to `1` indicates success.

The function returns the result of the method call in the form of a *VDM C++ Library* `Generic` object.

It is the user's responsibility to provide an implementation of this function.

5.4 The Function `DlClass_delete`

This function is called by the Toolbox whenever an instance of a DL Class is to be destroyed. The Toolbox interpreter uses a garbage collection scheme based on reference counting, so this function is called whenever the reference count for an instance of a DL Class reaches 0. The function is also called when objects are destroyed as a result of a dlclose call (see Section 5.9).

It is the user's responsibility to provide an implementation of this function.

5.5 The Function `DlClass_new`

This function is called by the Toolbox whenever an instance of a DL Class is created. It takes as parameter a string representing the name of the class being instantiated and returns a pointer to an instance of `DlClass`.

It is the user's responsibility to provide an implementation of this function.

5.6 The Class `DlClass`

The class `DlClass` is an abstract class specifying a prototype for one method: `DlMethodCall` which is used by a specific class to translate method calls at the VDM++ level into method calls on the external code. Note that indirection via `DlClass_call` is necessary because the interface called by the Toolbox interpreter must be C code.

It is the user's responsibility to provide an implementation of a subclass of this class for each external class which is being shadowed at the VDM++ level.

5.7 The DLObject Class

The `DLObject` is an extension to the *VDM C++ Library* which allows passing of objects between the Toolbox interpreter and the external code. Each `DLObject` corresponds to an instance of DL Class and contains `DlClass` pointer. The class has the following methods:

`DLObject(const string & name, DlClass *object)` Constructor that creates a `DLObject` of the DL Class called `name` which is initialized with a pointer to a `DlClass` given by `object`.

`DLObject(const DLObject & dlobj)` Constructor that creates a new `DLObject` from an existing one.

`DLObject(const Generic & dlobj)` Constructor used for narrowing a `Generic`.

`DLObject & operator=(const DLObject & dlobj)` Gives this object the value of `dlobj`.

`DLObject & operator=(const Generic & gen)` Gives this object the value of `gen`.

`string GetName() const` Returns the name of the VDM++ DL Class that this object corresponds to.

`DlClass * GetPtr() const` Returns the pointer to the external instance of `DlClass`.

`DLObject:: DLObject()` Destructor.

5.8 Opening Libraries

Whenever the Toolbox command init is used all shared libraries corresponding to DL Classes are opened. These remain open until either a dlclose command is issued (see Section 5.9) or init is called again. In the latter case the currently open libraries are all closed and then reopened as part of the initialization.

5.9 Closing Libraries

On windows, dlls can only be overwritten when they are not loaded by another application. For this purpose you can use the dlclose command in the interpreter window (or command prompt for the command line version of the Toolbox) to close the loaded dlls.

dlclose deletes the external C++ objects then closes the dlls. The VDM++ objects are not deleted in the Toolbox but they are marked as having no external C++ objects associated. If you then try to execute specifications that would invoke external methods on existing dlclass objects you will get a run time error. Non-dlclass method calls can still be done after a dlclose command, though.

Most likely you will have to use init after a dlclose and after you have copied the new dlls.

5.10 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 interface layer must fulfill the following requirements:

- The prototype C functions defined in `dlclass.h` must be implemented in the interface layer;
- Each class in the external code for which there is a corresponding DL Class should be wrapped as a subclass of `DlClass` and an implementation of the class method `DlMethodCall` must be provided.
- The `dlclass.h` header file, which is supplied as part of the distribution, must be included in the file(s) with the interface layer code since it contains prototypes of the type-specific functions of the *VDM C++ Library*.

See the example makefiles in Appendices D and E for the actual compiler flags to use.

A shared library should have the file extension `".so"` on Unix and `.dll` on Windows.

References

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

[SCSb] SCSK. *The VDM++ Language*. SCSK.

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

[SCSd] SCSK. *VDM++ 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 of the *VDM++ Toolbox* you need a Toolbox license which includes the Dynamic Link feature (a line with the `vppdl` feature must be present in the license file). The *VDM C++ Library* and a compiler for creating an executable shared library from the integrated code are also required.

This feature runs on the following combinations:

- 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

Note that shared libraries are *extremely* compiler sensitive. Deviation from the above combinations will most likely lead to runtime errors when using DL Classes.

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 dlclass.h

```
////////////////////////////////////////////////////////////////////////
// $Id: dlclass.h,v 1.4 2006/04/20 07:38:35 vdmtools Exp $
// dlclass.h
// Interface header file for VDM++ dlclass'es.
////////////////////////////////////////////////////////////////////////

#include "metaiv.h"

#ifndef _dlclass_h_
#define _dlclass_h_

#ifdef _MSC_VER
#define DLLFUN __declspec(dllexport)
#else
#define DLLFUN
#endif // _MSC_VER

#include <sstream>

class DlClass;

extern "C" {
  /**
   * This method is used to create a class.
   * @param name Name of the class as a String (plain class, no packages!)
   * @returns Class instance if successful, else 0
   */
  DLLFUN DlClass* DlClass_new (const wchar_t* name);

  /**
   * This method is used to delete a class
   * @param c C++ pointer to the class
   * @returns true, if successful, 0 if not
   * (anyway, we cannot do anything if an error occurs)
   */
  DLLFUN int DlClass_delete (DlClass* c);

  /**
   * Method used to call methods on a class.
```

```
   * @param c Pointer to the class instance
   * @param name name of the method to be called as a wstring.
   * @param params Parameters to the call as MetaIV-value
   * @success reference to int, that indicates if the call was
   * successful (boolean)
   */
  DLLFUN Generic DlClass_call (DlClass* c, const wchar_t * name, const Sequence &

// 20121005 -->
//  DLLFUN void DlClass_callp (DlClass* c, const wchar_t * name, const char * para
// <-- 20121005
}

/**
* Interface for the DLClass
* We could return void-pointers instead, but if
* all used DLClasses inherit from DlClass, the
* customer can use dynamic_cast
*/
class DlClass
{
public:
  virtual Generic DlMethodCall (const wchar_t * n, const Sequence & p) = 0;

  virtual ~DlClass() {};
};

#endif // _dlclass_h_
```

C Example Files

C.1 VDM++ Layer

```
dlclass BigInt

--uselib "bigint_dl.so" -- Unix
uselib "bigint_dl.dll"  -- Windows

operations

public Create : nat ==> BigInt
Create(n) ==
( SetVal(n);
  return self
);

protected SetVal : nat ==> ()
SetVal(n) ==
  is not yet specified;

public toString : () ==> seq of char
toString() ==
  is not yet specified;

public plus : BigInt ==> BigInt
plus(i) ==
  is not yet specified;

public minus : BigInt ==> BigInt
minus(i) ==
  is not yet specified;

public gt : BigInt ==> bool
gt(i) ==
  is not yet specified;

end BigInt

class BankAccount
```

```
instance variables
  name : seq of char;
  number : nat;
  balance : BigInt

operations

public Init : seq of char * nat ==> ()
Init(nname, nnum) ==
( name := nname;
  number := nnum;
  balance := new BigInt().Create(nnum);
);

public Deposit : BigInt ==> ()
Deposit(bi) ==
  balance := balance.plus(bi);

public Withdraw : BigInt ==> ()
Withdraw(bi) ==
  balance := balance.minus(bi)
pre balance.gt(bi);

public GetBalance : () ==> BigInt
GetBalance() ==
  return balance

end BankAccount

class A

operations

public Test : () ==> seq of char
Test() ==
( dcl b : BankAccount := new BankAccount();
  dcl bi : BigInt := new BigInt().Create(400);
  b.Init("customer", 100);

  b.Deposit(bi);
```

```
  return b.GetBalance().toString();
)

end A
```

C.2 bigint_dl.h

```
#ifndef _bigint_dl_h_
#define _bigint_dl_h_

#include "dlclass.h"
#include "m_apm.h"
#ifdef CG
#include "BigInt.h"
#endif //CG

class BigIntDL : public DlClass
{
  MAPM val;

 private:
#ifdef CG
  BigIntDL *GetDLPtr(const ObjectRef& obj);
#else
  BigIntDL *GetDLPtr(const DLObject& obj);
#endif //CG

 public:
  void setVal(MAPM);
  MAPM getVal();
  Generic DlMethodCall (const wchar_t* name, const Sequence &p);
  Generic SetVal (const Sequence &p);
#ifdef CG
  ObjectRef plus (const Sequence &p);
#else
  DLObject plus (const Sequence &p);
#endif //CG
  Sequence toString();
```

```
};

#endif //_bigint_dl_h_
```

C.3 bigint_dl.cc

```
#include "bigint_dl.h"

DlClass* DlClass_new (const wchar_t* name)
{
#ifdef DEBUG
  wcerr << L"DlClass_new called" << endl;
#endif //DEBUG
  if (!wcscmp(name, L"BigInt"))
    return new BigIntDL ();
  else
    return 0;
}

int DlClass_delete (DlClass* c)
{
#ifdef DEBUG
  wcerr << L"DlClass_delete called" << endl;
#endif //DEBUG
  try {
    delete c;
  } catch (...) {
    return 0;
  }
  return 1;
}

Generic DlClass_call (DlClass* c, const wchar_t* name, const Sequence& params, int
{
#ifdef DEBUG
  wcerr << L"DlClass_call: " << name << L"(" << params << L")" << endl;
#endif //DEBUG
  Generic result;
  try {
```

```
    result = c->DlMethodCall (name, params);
  } catch (...) {
    success = 0;
    return result;
  }
  success = 1;
  return result;
}

Generic BigIntDL::DlMethodCall (const wchar_t* name, const Sequence &p)
{
  Generic res;

  if (!wcscmp (name, L"SetVal"))
    res = this->SetVal(p);
  else if (!wcscmp(name, L"plus"))
    res = this->plus(p);
  else if (!wcscmp(name, L"toString"))
    res = this->toString();
  else {
      // the method does not exist
  }

  return res;
}

Generic BigIntDL::SetVal (const Sequence &p)
{
#ifdef DEBUG
  wcout << L"BigIntDL::SetVal" << endl;
#endif //DEBUG
  Int n(p.Hd());
  int nVal = n.GetValue();
  val = MAPM(nVal);
  return Generic();
}

#ifdef CG
BigIntDL *BigIntDL::GetDLPtr(const ObjectRef& obj)
#else
BigIntDL *BigIntDL::GetDLPtr(const DLObject& obj)
```

```
#endif //CG
{
#ifdef CG
  vdm_BigInt *objRefPtr = ObjGet_vdm_BigInt(obj);
  BigIntDL *objPtr = (BigIntDL*) objRefPtr->BigInt_dlClassPtr;
#else
  BigIntDL *objPtr = (BigIntDL*) obj.GetPtr();
#endif
  return objPtr;
}


#ifdef CG
ObjectRef BigIntDL::plus (const Sequence &p)
#else
DLObject BigIntDL::plus (const Sequence &p)
#endif //CG
{
#ifdef DEBUG
  wcout << L"BigIntDL::plus" << endl;
#endif //DEBUG
  // Extract arguments
  BigIntDL *argPtr = GetDLPtr(p.Hd());

  // Set up result object
#ifdef CG
  ObjectRef result (new vdm_BigInt);
#else
  DLObject result(L"BigInt", new BigIntDL);
#endif

  BigIntDL *resPtr = GetDLPtr(result);

  // Perform manipulation on pointers, as needed for function
  resPtr->setVal( val + argPtr->getVal());

  return result;
}

Sequence BigIntDL::toString()
{
```

```
#ifdef DEBUG
  wcout << L"BigIntDL::toString" << endl;
#endif //DEBUG
  char res[100];
  val.toIntegerString(res);
  return Sequence(string2wstring(string(res)));
}

void BigIntDL::setVal(MAPM newVal)
{
  val = newVal;
}

MAPM BigIntDL::getVal()
{
  return val;
}
```

D Unix Makefile

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

OSTYPE=$(shell uname)

MAPMDIR = ./mapm_3.60
INCL    = -I../../cg/include -I${MAPMDIR}

VPPDE = ../../bin/vppde
CXXFLAGS = $(INCL)
DEBUG =-g
VDMLIB  = ../../cg/lib
CGFLAGS = -DCG #-DDEBUG

ifeq ($(strip $(OSTYPE)),Darwin)
OSV = $(shell uname -r)
OSMV = $(word 1, $(subst ., ,$(strip $(OSV))))
CCPATH = /usr/bin/
ifeq ($(strip $(OSMV)),12) # 10.8
CC      = $(CCPATH)clang++
CXX     = $(CCPATH)clang++
else
ifeq ($(strip $(OSMV)),11) # 10.7
CC      = $(CCPATH)clang++
CXX     = $(CCPATH)clang++
else
ifeq ($(strip $(OSMV)),10) # 10.6
CC      = $(CCPATH)g++
CXX     = $(CCPATH)g++
else
ifeq ($(strip $(OSMV)),9) # 10.5
CC      = $(CCPATH)g++-4.2
CXX     = $(CCPATH)g++-4.2
else
ifeq ($(strip $(OSMV)),8) # 10.4
CC      = $(CCPATH)g++-4.0
CXX     = $(CCPATH)g++-4.0
else
```

```
CC      = $(CCPATH)g++
CXX     = $(CCPATH)g++
endif
endif
endif
endif
endif

LIB     = -L. -L$(VDMLIB) -lCG -lvdm -lm -liconv

#ifeq ($(strip $(GCC_VERSION)),-3.3)
#MAPMLIB = lib_mapm_darwin_ppcg3.a
#else
MAPMLIB = lib_mapm_darwin.a
#endif
endif

ifeq ($(strip $(OSTYPE)),Linux)
CPUTYPE = $(shell uname -m)
CC      = /usr/bin/g++
CXX     = /usr/bin/g++
LIB     = -L. -L$(VDMLIB) -lCG -lvdm -lm
ifeq ($(strip $(CPUTYPE)),x86_64)
MAPMLIB = lib_mapm_linux_x86_64.a
else
MAPMLIB = lib_mapm.a
endif
endif

ifeq ($(strip $(OSTYPE)),SunOS)
CC      = /usr/sfw/bin/g++
CXX     = /usr/sfw/bin/g++
LIB     = -L. -L$(VDMLIB) -L../../lib -lCG -lvdm -lm
MAPMLIB = lib_mapm_solaris.a
endif

ifeq ($(strip $(OSTYPE)),FreeBSD)
CC      = /usr/bin/g++
CXX     = /usr/bin/g++
LIB     = -L. -L$(VDMLIB) -L../../lib -lCG -lvdm -lm -L/usr/local/lib -liconv
MAPMLIB = lib_mapm_freebsd.a
```

```
endif

OSTYPE2=$(word 1, $(subst _, ,$(strip $(OSTYPE))))
ifeq ($(strip $(OSTYPE2)),CYGWIN)
all: winall
else
all: cg.stamp
        $(MAKE) cgex bigint_dl.so
endif
        $(MAKE) CGEX

CGSOURCES = A.cc A_anonym.cc BankAccount.cc BankAccount_anonym.cc \
            BigInt.cc BigInt_anonym.cc CGBase.cc
CGOBJS = $(CGSOURCES:%.cc=%.o)

winall:
        make -f Makefile.win32

cgex:   cgex.o $(CGOBJS) bigint_cg.o $(MAPMDIR)/$(MAPMLIB)
ifeq ($(strip $(OSTYPE)),Darwin)
        ranlib $(MAPMDIR)/$(MAPMLIB)
endif
        $(CC) $(DEBUG) $(INCL) $(CGFLAGS) -o $@ $^ $(LIB)

cgex.o: cgex.cc A_userdef.h BankAccount_userdef.h cg.stamp
        $(CC) $(DEBUG) $(INCL) $(CGFLAGS) -c -o $@ cgex.cc

%.o: %.cc
        $(CC) $(DEBUG) $(CCFLAGS) $(INCL) -c -o $@ $^

%_shared.o: %.cc
        $(CC) $(DEBUG) -fPIC $(CCFLAGS) $(INCL) -c -o $@ $^

cg.stamp: bigint.vpp
        $(VPPDE) -c $^
        touch $@

jcg.stamp: bigint.vpp
        $(VPPDE) -j $^
        touch $@
```

```
%_userdef.h:
        touch $@

bigint_dl.so: bigint_dl.o ${MAPMDIR}/$(MAPMLIB)

bigint_dl.o: bigint_dl.cc
        $(CC) $(DEBUG) -fPIC $(CCFLAGS) $(INCL) -c -o $@ $^

bigint_cg.o: bigint_dl.cc
        $(CC) $(DEBUG) $(INCL) $(CGFLAGS) -c -o $@ $^

%.so:
ifeq ($(strip $(OSTYPE)),Darwin)
        ranlib ${MAPMDIR}/$(MAPMLIB)
        $(CXX) -dynamiclib -fPIC -o $@ $^ $(LIB)
else
        $(CXX) -shared -fPIC -o $@ $^ $(LIB)
endif

GENJAVAFILES = A.java BankAccount.java BigInt.java

CGEX.class: CGEX.java $(GENJAVAFILES)
CGEX: jcg.stamp CGEX.class $(GENJAVAFILES:%.java=%.class)

ifeq ($(strip $(OSTYPE)),Darwin)
%.class : %.java
        javac -J-Dfile.encoding=UTF-8 -encoding UTF8 -classpath ../../javacg/VDM.j
else
ifeq ($(strip $(OSTYPE2)),CYGWIN)
%.class : %.java
        javac -classpath "../../javacg/VDM.jar;." $<
else
%.class : %.java
        javac -classpath ../../javacg/VDM.jar:. $<
endif
endif

ifeq ($(strip $(OSTYPE)),Darwin)
jrun:   jcg.stamp CGEX
        java -Dfile.encoding=UTF-8 -classpath ../../javacg/VDM.jar:. CGEX
else
```

```
ifeq ($(strip $(OSTYPE2)),CYGWIN)
jrun:   jcg.stamp CGEX
        java -classpath "../../javacg/VDM.jar;." CGEX
else
jrun:   jcg.stamp CGEX
        java -classpath ../../javacg/VDM.jar:. CGEX
endif
endif

clean:
        rm -f bigint_dl.so bigint_dl.o bigint_cg.o
        rm -f cgex cgex.o cg.stamp $(CGOBJS) $(CGOBJS:%.o=%_shared.o)
        rm -f $(CGSOURCES)
        rm -f A.h A_anonym.h A_userdef.h
        rm -f BankAccount.h BankAccount_anonym.h BankAccount_userdef.h
        rm -f BigInt.h BigInt_anonym.h
        rm -f CGBase.h
        rm -f *.obj *.cpp *.exe *.exp *.lib
        rm -f A.java BankAccount.java BigInt.java *.class *.java.bak jcg.stamp
```

E Windows Makefile

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

TBDIR = ../..
WTBDIR = ../..

#TBDIR = /cygdrive/c/Program Files/The VDM++ Toolbox v2.8
#WTBDIR = C:/Program Files/The VDM++ Toolbox v2.8

VPPDE = "$(TBDIR)/bin/vppde"

CC      = cl.exe
MAPMDIR = ./mapm_3.60
CGLIBS = "$(WTBDIR)/cg/lib/CG.lib" "$(WTBDIR)/cg/lib/vdm.lib"

CFLAGS   = /nologo /c /MD /W0 /EHsc /D "WIN32" /TP
INCPATH  = -I"$(WTBDIR)/cg/include" -I./mapm_3.60

LINK     = link.exe
LPATH    = /LIBPATH:"$(WTBDIR)/cg/lib"

WINMTLIBS = libcpmt.lib libcmt.lib
WINMDLIBS = msvcrt.lib msvcprt.lib
WINLIBS  = $(WINMDLIBS) \
        kernel32.lib user32.lib gdi32.lib comdlg32.lib \
        advapi32.lib shell32.lib uuid.lib winspool.lib \
        oldnames.lib comctl32.lib

LDFLAGS = /nologo /NODEFAULTLIB "$(MAPMDIR)/mapm.lib" $(CGLIBS) $(WINLIBS)

## DLL Specific binary, path and flag
DLLFLAGS = /D "_USRDLL"
DLL_LFLAGS = /nologo /dll /incremental:no /NODEFAULTLIB $(CGLIBS) #/DEBUG
DLLWINLIBS  = $(WINLIBS)

## CG Version Files
```

```
CGSOURCES = A.cpp A_anonym.cpp BankAccount.cpp BankAccount_anonym.cpp \
            BigInt.cpp BigInt_anonym.cpp CGBase.cpp
CGOBJ = A.obj A_anonym.obj BankAccount.obj BankAccount_anonym.obj BigInt.obj \
        BigInt_anonym.obj CGBase.obj
CGFLAGS = /D CG #-DDEBUG

## CG Version specific rules

all: cgex.exe bigint_dl.dll

cgex.obj: cg.stamp A_userdef.h BankAccount_userdef.h

cg.stamp: bigint.vpp
        $(VPPDE) <cg.script
        echo  >cg.stamp

cgex.exe: cgex.obj bigint_cg.obj $(CGOBJ)

bigint_cg.obj: bigint_dl.cpp
        $(CC) $(CFLAGS) $(CGFLAGS) $(INCPATH) bigint_dl.cpp -Fo"bigint_cg.obj"

A.obj: A.cpp
A_anonym.obj: A_anonym.cpp
BigInt.obj: BigInt.cpp
BigInt_anonym.obj: BigInt_anonym.cpp
BankAccount.obj: BankAccount.cpp
BankAccount_anonym.obj: BankAccount_anonym.cpp
CGBase.obj: CGBase.cpp

## DLL specific  rules

bigint_dl.obj: bigint_dl.cpp
        $(CC) $(CFLAGS) $(DLLFLAGS) $(INCPATH) /Fo"$@" $<

bigint_dl.dll: bigint_dl.obj $(MAPMDIR)/mapm.lib

## Rules

%.dll:
        $(LINK) $(DLL_LFLAGS) /out:"$@" $^  $(LPATH) vdm.lib $(DLLWINLIBS)
```

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

%.exe: %.obj
        $(LINK) /OUT:$@ $^ $(LDFLAGS)

%.cpp: %.cc
        cp -f $^ $@

%_userdef.h:
        touch $@

clean:
        rm -f cg.stamp
        rm -f $(CGOBJ) bigint_cg.obj cgex.obj cgexe.exe
        rm -f bigint_dl.dll bigint_dl.obj
        rm -f *.cpp
        rm -f A.h A_anonym.h A_userdef.h
        rm -f BankAccount.h BankAccount_anonym.h BankAccount_userdef.h
        rm -f BigInt.h BigInt_anonym.h
        rm -f CGBase.h
        rm -f cgex.exe cgex.obj cgex.lib cgex.exp
        rm -f bigint_dl.lib bigint_dl.exp  bigint_dl.pdb
```