九州大学
KYUSHU UNIVERSITY

VDMTools

The VDM C++ Library

ver.1.0

How to contact:

http://fmvdm.org/	VDM information web site(in Japanese)
http://fmvdm.org/tools/vdmtools	VDMTools web site(in Japanese)
inq@fmvdm.org	Mail

The VDM C++ Library 1.0
— Revised for VDMTools v9.0.6

Contents

1 Introduction

This document contains a description of the classes and methods which constitute the VDM C++ library. Some knowledge of C++ is a prerequisite in order to read this document. For each VDM type a corresponding C++ class exists implementing this type. In addition for the compound types sets, maps and sequences templates exist in the VDM C++ library that makes it possible to declare types with better type information.

Section 2 lists the notational conventions used in this document. In section 3 the general structure of a VDM object is briefly presented. In section 4 functions which are common to all VDM classes are described whereas section 5 lists the specific functions which can be performed on the different VDM classes. The templates for sets, maps and sequences are described in Section 6. All error messages are described in section 7.

In appendix A the files which constitute the VDM C++ Library are listed.

The current version of the library can be used with:

- 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

2 Notation Conventions

The following conventions will be used in this document:

Variable Name	*Variable Type*	*C++ Class*
`i`	C++ int	`int`
`c`	C++ char	`char`
`d`	C++ double	`double`
`s`	C++ string	`string`
`I`	VDM Integer	`Int`
`M`	VDM Map	`Map`

Variable Name	*Variable Type*	*C++ Class*
C	VDM Char	Char
B	VDM Bool	Bool
N	VDM Nil	Nil
Q	VDM Quote	Quote
G	VDM Generic	Generic
Rl	VDM Real	Real
Rc	VDM Record	Record
Tx	VDM Text	Text
Tp	VDM Tuple	Tuple
Tk	VDM Token	Token
St	VDM Set	Set
Sq	VDM Sequence	Sequence
Ob	VDM Object Reference	ObjectRef
A	Any of the above described VDM types	

3 The general structure of a VDM value

This section will briefly describe the general structure of a VDM value in order to give an idea of what happens "under the surface". It is important to have some basic knowledge of this in order to use the VDM C++ classes.

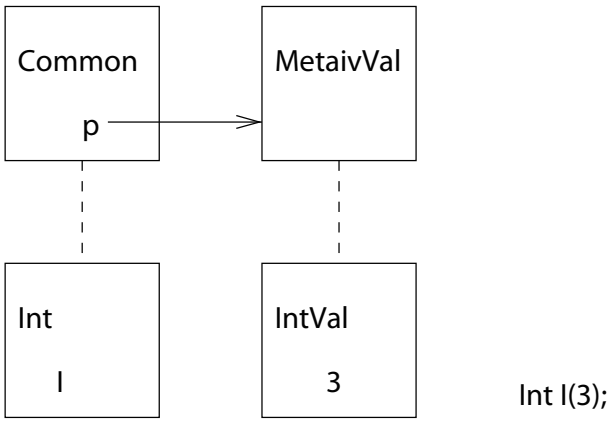

Figure 1: General structure.

For all the VDM classes like `Set`, `Int`, `Map` etc. there exists a corresponding value class named `SetVal`, `IntVal`, `MapVal` etc. All the VDM classes are subclasses of the class `Common`, whereas all the value classes are subclasses of the class `MetaivVal`.

When for instance a variable `I` is declared of type `Int`, instances of types `Int` and `IntVal` are created and a pointer `p` from the `Int` instance (the pointer is actually defined in `Common`) is pointing to the `IntVal` instance (it is actually pointing to the `MetaivVal` part of the `IntVal` instance). This situation is illustrated in Figure 1. The dashed lines illustrate the class hierarchy and the solid line illustrates the pointer `p`. The value of `I` is located in the `IntVal` instance. From now on when we refer to the value of a variable we always mean the instance of the Val class which the pointer `p` points to.

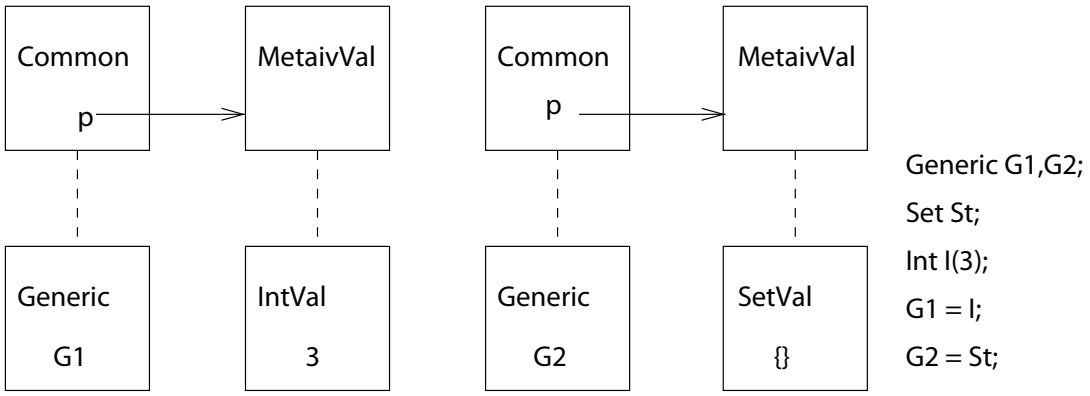

Figure 2: Generics.

In order to support the idea of union types the `Generic` class has been introduced. This allows instances of the classes representing compound VDM types (`Map, Sequence, Tuple, Set, Record`) to contain elements of different types at the same time. The value of a generic can be any of the VDM values from the basics `IntVal, RealVal` to the compound values like `TupleVal, SequenceVal`. This means that a variable of type `Generic` can have an underlying value of any VDM type. Figure 2 shows two examples of generics.

In reality it is implemented such that the compound types only can contain elements of type `Generic` Most of the functions which insert elements into compound types are able to automatically cast elements to `Generics` before inserting them, but functions which retrieve elements will always return a `Generic`.

To show an example of this let us look at the VDM expression:

```
let Sq = [1,<TWO>] in
  ...
```

The sequence `Sq` contains both an integer and a quote. As C++ is strongly typed the C++ implementation of the let-expression must cast the integer and the quote to `Generic` before appending them to the sequence `Sq`. In the implementation below this casting is done automatically by the `ImpAppend` function, and the retrieve function `Hd()` returns a `Generic`. As the first element in the sequence is of type `Int`, `G` can later be cast to `Int`.

```
Sequence Sq;  Generic G;

Sq.ImpAppend(Int(1));
Sq.ImpAppend(Quote(``<TWO>''));

G = Sq.Hd();
```

In general, any type can be casted to a `Generic`, but it will only be generic on the surface and preserves the underlying value. For instance, the generic `G1` on Figure 2 can later be casted back to an `Int`. For all the classes representing compound types (`Map, Sequence, Tuple, Set, Record`) all the elements contained in the class will automatically be casted to `Generic` before they are included. This means that when an element is retrieved from a compound variable it will always be of type `Generic`. It can then be casted back to the original type if necessary.

4 General functions on VDM types

This section describes all functions which are defined in class `Common` and therefore are applicable to instances of all types. Note that `true` in the following denotes an integer greater than zero and `false` denotes the integer value zero.

Functions

`A.MyValType()`

Returns the type of the value of `A`.

Result type : **metaivType**

Note that the enumerate type `metaivType` is a part of the library.

```
enum metaivType {
  mt_nil, mt_char, mt_int, mt_real, mt_quote,
  mt_tuple, mt_record, mt_set, mt_map, mt_generic,
  mt_text, mt_token, mt_bool, mt_sequence,
  mt_objectref, mt_undef
}
```

`A1 = A2`

Gives `A1` the value of `A2`. If `A1` is of type `Generic` then `A2` can be of any type. In that case `A1` will still be a generic, but it will contain the value of `A2`. Otherwise the type of the value of `A2` must be the same as the type of the value of `A1`.

Result type : reference to A1.

`A1 == A2`

Returns `true` if the value of `A1` equals the value of `A2` and `false` otherwise.

Result type : **bool**

`A1 != A2`

Returns `true` if the value of `A1` does not equal the value of `A2` and `false` otherwise.

Result type : **bool**

`A.ascii()`

Returns a `wstring` containing an ASCII representation of the VDM value.

Result type : **wstring**

`A.IsNil()`

Returns `true` if the value of `A` is of type `Nil`

Result type : **bool**

`A.IsChar()`

Returns `true` if `A` is of type `Char`

Result type : **bool**

`A.IsInt()`

Returns `true` if `A` is of type `Int`

Result type : **bool**

`A.IsReal()`
Returns `true` if `A` is of type `Real`
Result type : **bool**

`A.IsQuote()`
Returns `true` if `A` is of type `Quote`
Result type : **bool**

`A.IsTuple()`
Returns `true` if `A` is of type `Tuple`
Result type : **bool**

`A.IsRecord()`
Returns `true` if `A` is of type `Record`
Result type : **bool**

`A.IsSet()`
Returns `true` if `A` is of type `Set`
Result type : **bool**

`A.IsMap()`
Returns `true` if `A` is of type `Map`
Result type : **bool**

`A.IsText()`
Returns `true` if `A` is of type `Text`
Result type : **bool**

`A.IsToken()`
Returns `true` if `A` is of type `Token`
Result type : **bool**

`A.IsBool()`
Returns `true` if `A` is of type `Bool`
Result type : **bool**

`A.IsSequence()`
Returns `true` if `A` is of type `Sequence`
Result type : **bool**

`A.IsObjectRef()`
: Returns `true` if `A` is of type `ObjectRef`

  Result type : **bool**

`A.WriteVal(o)`
: Write the value of `A` to the `ostream` `o`, in a format such that the value can be read in again using the function ReadVal. `ReadVal` and `WriteVal` is used for saving a value to the filesystem and later read it back ("persistency").

  Return type: **void**

`Generic g; g = ReadVal(i)`
: Read in a value (through an istream) from a file that was written with the WriteVal method. Note that `ReadVal` is a function, not a method.

  The format that `WriteVal` writes and `ReadVal` reads is a low-level format not intended for humans.

  Return type: **Generic**

Examples

Example C++ program using the functions described in this section.

```
Generic G1;  Int I1(3),I2(4),I3;

G1 = I1;
I3 = G1;
if (G1 == I3)
  wcout << "The value of G1 equals the value of I3" << endl;
if (G1.IsInt())
  wcout << "The value of G1 is of type Int" << endl;
  wcout << G1.MyValType() << "  The value of G1 is of type Int" << endl;
  wcout << I1.ascii()  << "  The ASCII representation of I1" << endl;
```

The result of running this program is :

```
The value of G1 equals the value of I3
The value of G1 is of type Int
103  The value of G1 is of type Int
3  The ASCII representation of I1
```

4.1 Printing values to ostreams

Any VDM value `v` can be printed to an ostream `os` with `os << v`.

A Record will per default be printed with its numeric tag like this: `mk_unknown4(...)`.

Example

```
#include "metaiv.h"

int main(int, char**)
{
VDMGetDefaultRecInfoMap().NewTag(10, 1);
VDMGetDefaultRecInfoMap().SetSymTag(10, L"X‘a");
VDMGetDefaultRecInfoMap().NewTag(11, 1);
VDMGetDefaultRecInfoMap().SetSymTag(11, L"X‘b");

Record r1(10,1);
Record r2(11,1);
Record r3(11,1);
r3.SetField(1, Int(100));
r2.SetField(1, r3);
r1.SetField(1, r2);
wcout << "r1=" << r1 << endl;
}
```

The result of running this program is :

```
r1=mk_X‘a(
mk_X‘b(
mk_X‘b( 100 ) ) )
```

5 Specific functions on the VDM types

5.1 Int

`Int` supports the following member functions:

`Int I`
: Declares `I` as an `Int`. The value of `I` is initialized to 0.

Result type : **void**

`Int I(i)`
: Declares `I` as an `Int` and initializes the value to `i`.

Result type : **void**

`Int I(I1)`
: Declares `I` as an `Int` which is equal to `I1`.

Result type : **void**

`I.GetValue()`
: Returns the C++ integer value of `I`.

Result type : **int**

`I = i`
: Gives `I` the value of `i`.

Result type : **Int&**

`-I`
: Unary minus. Return an `Int` with the negated value of `I`.

Result type : **Int**

`I1 + I2`
: Binary plus.

Result type : **Int**

`I + R`
: Binary plus.

Result type : **Real**

`I1 - I2`
: Binary minus.

  Result type : **Int**

`I - R`
: Binary minus.

  Result type : **Real**

`I1 * I2`
: Binary multiplication.

  Result type : **Int**

`I * R`
: Binary multiplication.

  Result type : **Real**

`I1 / I2`
: Binary division.

  Division by zero causes an error cf. Section 7.

  Result type : **Real**

`I / R`
: Binary division.

  Division by zero causes an error cf. Section 7.

  Result type : **Real**

`I1.Exp(I2)`
: Exponentiation.

  Result type : **Real**

`I.Exp(R)`
: Exponentiation.

  Result type : **Real**

Examples

```
Int I1(3),I2;
Int I3(I1);
```

```
    if (I1 == I3)
      wcout << "The value of I1 equals the value of I3" << endl;
      wcout << I2.GetValue() << "  is the initial value of I2" << endl;
    I2 = 10;
      wcout << I2.GetValue() << "  is the new value of I2" << endl;
```

The result of running this program is :

```
The value of I1 equals the value of I3
0  is the initial value of I2
10  is the new value of I2
```

5.2 Real

`Real` supports the following member functions:

`Real Rl`
Declares `Rl` as a `Real`. The value of `Rl` is initialized to 0.

Result type : **void**

`Real Rl(d)`
Declares `Rl` as a `Real` and initializes the value to `d`.

Result type : **void**

`Real Rl(Rl1)`
Declares `Rl` as a `Real` which is equal to `Rl1`.

Result type : **void**

`Rl.GetValue()`
Returns the C++ double value of `Rl`.

Result type : **double**

`Rl = d`
Gives `Rl` the value of `d`.

Result type : **Real&**

`-R`
: Unary minus. Returns a `Real` with the negated value of `R`.

Result type : **Real**

`R1 + R2`
: Binary plus.

Result type : **Real**

`R + I`
: Binary plus.

Result type : **Real**

`R1 - R2`
: Binary minus.

Result type : **Real**

`R - I`
: Binary minus.

Result type : **Real**

`R1 * R2`
: Binary multiplication.

Result type : **Real**

`R * I`
: Binary multiplication.

Result type : **Real**

`R1 / R2`
: Binary division.

Result type : **Real**

`R / I`
: Binary division.

Result type : **Real**

`R1.Exp(R2)`
: Exponentiation.

Result type : **Real**

`R.Exp(I)`
: Exponentiation.

Result type : **Real**

Examples

```
Real Rl1(3.2),Rl2;
Real Rl3(Rl1);

if (Rl1 == Rl3)
  wcout << "The value of Rl1 equals the value of Rl3" << endl;
  wcout << Rl2.GetValue() << "  is the initial value of Rl2" << endl;
Rl2 = 10.5;
  wcout << Rl2.GetValue() << "  is the new value of Rl2" << endl;
```

The result of running this program is :

```
The value of Rl1 equals the value of Rl3
0 is the initial value of Rl2
10.5  is the new value of Rl2
```

5.3 Bool

`Bool` supports the following member functions:

`Bool B`
: Declares `B` as a `Bool`. The value of `B` is initialized to 0 (`false`).

  Result type : **void**

`Bool B(i)`
: Declares `B` as a `Bool` and initialize the value to `i`.

  Result type : **void**

`Bool B(B1)`
: Declares `B` as a `Bool` which is equal to `B1`.

  Result type : **void**

`B.GetValue()`
: Returns the C++ **bool** of `B`.

  Result type : **bool**

`B = i`
: Gives `B` the value of `i`.

Result type : **Bool&**

`B.not()`
: Logical negation.

Result type : **Bool**

`!B`
: Logical negation.

Result type : **Bool**

`B1.and(B2)`
: Logical and.

Result type : **Bool**

`B1 && B2`
: Logical and.

Result type : **Bool**

`B1.or(B2)`
: Logical or.

Result type : **Bool**

`B1 || B2`
: Logical or.

Result type : **Bool**

Examples

```
Bool B1(3),B2;
Bool B3(B1);

if (B1 == B3)
  wcout << "The value of B1 equals the value of B3" << endl;
  wcout << B2.GetValue() << "  is the initial value of B2" << endl;
B2 = 1;
  wcout << B2.GetValue() << "  is the new value of B2 (true)" << endl;
```

The result of running this program is :

```
The value of B1 equals the value of B3
0  is the initial value of B2
1  is the new value of B2 (true)
```

5.4 Nil

`Nil` support the following member functions:

`Nil N`
Declares `N` as a value of type `Nil`.
Result type : **void**

Instances of type `Nil` only support the functions which are common to all types (see section 4).

5.5 Quote

`Quote` supports the following member functions:

`Quote Q`
Declares `Q` as a value of type `Quote`. The value of `Q` is initialized to `""`.
Result type : **void**

`Quote Q(s)`
Declares `Q` as a value of type `Quote` and initialize the value to `s`.
Result type : **void**

`Quote Q(Q1)`
Declares `Q` as a value of type `Quote` which is equal to `Q1`.
Result type : **void**

`Q.GetValue()`
Returns the C++ wstring value of `Q`.
Result type : **wstring**

`Q = s`
Gives `Q` the value of `s`.
Result type : **Quote&**

Examples

```
Quote Q1("Q_ONE"),Q2;
Quote Q3(Q1);

if (Q1 == Q3)
  wcout << "The value of Q1 equals the value of Q3" << endl;
  wcout << Q2.GetValue() << "  is the initial value of Q2" << endl;
Q2 = "Q_TWO";
  wcout << Q2.GetValue() << "  is the new value of Q2" << endl;
```

The result of running this program is :

```
The value of Q1 equals the value of Q3
  is the initial value of Q2
Q_TWO  is the new value of Q2
```

5.6 Char

`Char` supports the following member functions:

`Char C`
Declares `C` as a value of type `Char`. The value of `C` is initialized to `'?'`.
Result type : **void**

`Char C(c)`
Declares `C` as a value of type `Char` and initializes the value to `c`.
Result type : **void**

`Char C(C1)`
Declares `C` as a value of type `Char` which is equal to `C1`.

Result type : **void**

`C.GetValue()`
Returns the C++ wchar_t value of `C`.

Result type : **wchar_t**

`C = c`
Gives `C` the value of `c`.

Result type : **Char&**

Examples

```
Char C1('c'),C2;
Char C3(C1);

if (C1 == C3)
  wcout << "The value of C1 equals the value of C3" << endl;
  wcout << C2.GetValue() << "  is the initial value of C2" endl;
C2 = 'd';
  wcout << C2.GetValue() << "  is the new value of C2" << endl;
```

The result of running this program is :

```
The value of C1 equals the value of C3
?  is the initial value of C2
d  is the new value of C2
```

5.7 Text

`Text` supports the following member functions:

`Text Tx`
Declares `Tx` as a value of type `Text`. The value of `Tx` is initialized to `""`.

Result type : **void**

`Text Tx(s)`
Declares `Tx` as a value of type `Text` and initializes the value to `s`.

Result type : **void**

`Text Tx(Tx1)`
Declares `Tx` as a value of type `Text` which is equal to `Tx1`.

Result type : **void**

`Tx.GetValue()`
Returns the C++ wstring value of `Tx`.

Result type : **wstring**

`Tx = s`
Gives `Tx` the value of `s`.

Result type : **Text&**

Examples

```
Text Tx1("Tx_ONE"),Tx2;
Text Tx3(Tx1);

if (Tx1 == Tx3)
  wcout << "The value of Tx1 equals the value of Tx3" << endl;
  wcout << Tx2.GetValue() << "  is the initial value of Tx2" <<endl;
Tx2 = "Tx_TWO";
  wcout << Tx2.GetValue() << "  is the new value of Tx2" << endl;
```

The result of running this program is :

```
The value of Tx1 equals the value of Tx3
  is the initial value of Tx2
Tx_TWO  is the new value of Tx2
```

5.8 Token

`Token` supports the following member functions:

`Token Tk`
: Declares `Tk` as a value of type `Token`. The value of `Tk` is initialized to `""`.

  Result type : **void**

`Token Tk(s)`
: Declares `Tk` as a value of type `Token` and initializes the value to `s`.

  Result type : **void**

`Token Tk(Tk1)`
: Declares `Tk` as a value of type `Token` which is equal to `Tk1`.

  Result type : **void**

`Tk.GetValue()`
: Returns the C++ wstring value of `Tk`.

  Result type : **wstring**

`Tk = s`
: Gives `Tk` the value of `s`.

  Result type : **Token&**

Examples

```
Token Tk1("Tk_ONE"),Tk2;
Token Tk3(Tk1);

if (Tk1 == Tk3)
  wcout << "The value of Tk1 equals the value of Tk3" << endl;
  wcout << Tk2.GetValue() << "  is the initial value of Tk2" << endl;
Tk2 = "Tk_TWO";
  wcout << Tk2.GetValue() << "  is the new value of Tk2" <<endl;
```

The result of running this program is :

```
The value of Tk1 equals the value of Tk3
  is the initial value of Tk2
Tk_TWO  is the new value of Tk2
```

5.9 Map

`Map` supports the following member functions:

`Map M`
: Declares `M` as a value of type `Map` and initializes it to the empty map.

Result type : **void**

`Map M(M1)`
: Declares `M` as a value of type `Map` and initializes it to `M1`.

Result type : **void**

`M.Insert(A1, A2)`
: Inserts the key `A1` with the associated contents `A2` in `M`. If the key `A1` already belongs to the domain of `M` it is checked whether A2 is equal to the range value. If not, then an error is signaled cf. Section 7. The function returns a reference to `M`.

Result type : **Map&**

`M.ImpModify(A1, A2)`
: Works as `Insert` except that if the key `A1` already belongs to the domain of `M` the range value is modified to `A2`. The function returns a reference to `M`.

Result type : **Map&**

`M[A]`
: Returns the contents associated with the key `A`. If `A` is not in the domain for `M` an error is signaled cf. Section 7.

Result type : **Generic&**

`M.ImpOverride(M1)`
: `M` becomes the union of `M` and `M1`. If a key in `M` also exists in `M1` the corresponding contents in `M` is overriden by the contents in `M1`. The function returns a reference to `M`.

Result type : **Map&**

`M.Size()`
: Returns an integer denoting the number of keys in `M`.

Result type : **int**

`M.IsEmpty()`
: Returns `true` if `M.Size() == 0` and `false` otherwise.

  Result type : **bool**

`M.Dom()`
: Returns a Set containing all the keys in `M`.

  Result type : **Set**

`M.Rng()`
: Returns a Set containing all the contents in `M`.

  Result type : **Set**

`M.DomExists(A)`
: Returns `true` if `A` is in the domain of `M` and `false` otherwise.

  Result type : **bool**

`M.RemElem(A)`
: If `A` is in the domain of `M` remove both key and contents element, otherwise an error is signaled cf. Section 7.

  Result type : **Map&**

`M.First(G)`
: Returns 1 if `M` is not empty and returns the first key in the reference parameter `G`, according to an internal ordering. If `M` is empty it returns 0 and G will be returned as an empty Generic.

  Result type : **bool**

`M.Next(G)`
: Returns 1 and the next key in the reference parameter `G`. If there are no more keys in `M` it returns 0 and G will be returned as an empty Generic.

  Result type : **bool**

Examples

```
Map M1,M2;  Set St;  Int I(5);  Generic G;

M1.Insert(I,St).Insert(Int(7),St);
M2.Insert(I,Int(1));
  wcout << M1.ascii() << " is the value of M1 before overriding" << endl;
M1.ImpOverride(M2);
```

```
    wcout << M1.ascii() << " is the value of M1 after overriding" << endl;
    wcout << M1.Size() << " is the size of M1" << endl;
    wcout << M1.Dom().ascii() << " is the domain of M1" << endl;

  for (int b = M1.First(G); b; b = M1.Next(G))
    wcout << G.ascii() << " is a key of M1" << endl;
```

The result of running this program is :

```
{ 5 |-> {  }, 7 |-> {  } } is the value of M1 before overriding
{ 5 |-> 1, 7 |-> {  } } is the value of M1 after overriding
2 is the size of M1
{ 5, 7 } is the domain of M1
5 is a key of M1
7 is a key of M1
```

5.10 Sequence

Elements in a `Sequence` are indexed from 1 to the length of the sequence. `Sequence` supports the following member functions:

`Sequence Sq`
: Declares `Sq` as a value of type `Sequence` and initializes it to the empty sequence.

  Result type : **void**

`Sequence Sq(Sq1)`
: Declares `Sq` as a value of type `Sequence` and initializes it to `Sq1`.

  Result type : **void**

`Sequence Sq(s)`
: Declares `Sq` as a value of type `Sequence` and initializes it to the `wstring s` converted to a "`seq of char`".

  Result type : **void**

`Sq[i]`
: Returns the `i`'th element in `Sq`. If `i` is not a valid index, an error is signaled cf. Section 7.

Result type : **Generic&**

`Sq.Index(i)`
Returns the `i`'th element in `Sq`. If `i` is not a valid index, an error is signaled cf. Section 7.

Result type : **Generic&**

`Sq.Hd()`
Returns the first element in `Sq`. If `Sq` is the empty sequence, an error is signaled cf. Section 7.

Result type : **Generic&**

`Sq.Tl()`
Returns the tail of `Sq`. If `Sq` is the empty sequence, an error is signaled cf. Section 7.

Result type : **Sequence**

`Sq.ImpTl()`
Changes `Sq` to the tail of `Sq`. If `Sq` is the empty sequence, an error is signaled cf. Section 7.

The function returns a reference to `Sq`.

Result type : **Sequence&**

`Sq.RemElem(int i)`
Removed the `i`'th element from `Sq` given that `i` is a valid index for `Sq`. If it is not, an error is signaled cf. Section 7.

The function returns a reference to `Sq`.

Result type: **Sequence&**

`Sq.Length()`
Returns an integer denoting the number of elements in `Sq`.

Result type : **int**

`Sq.GetString(string& str)`
Convert a "`seq of char`" to `wstring`.

Returns `true` if all elements in `Sq` is of type Char and the string representation of `Sq` in the parameter `str`. Otherwise `GetString` will return `false` and set `str` to the empty string (`""`).

Result type : **bool**

`Sq.IsEmpty()`
Returns 0 if `Sq` is empty and 1 otherwise.

Result type : **bool**

`Sq.ImpAppend(A)`
Appends `A` to `Sq` and places the result in `Sq`. The function returns a reference to `Sq`.

Result type : **Sequence&**

`Sq.ImpModify(i,A)`
Modifies the `i`'th element in `Sq` to `A`. If `i` is not a valid index an error is signaled cf. Section 7. The function returns a reference to `Sq`.

Result type : **Sequence&**

`Sq.ImpPrepend(A)`
Prepends `A` to `Sq` and places the result in `Sq`. The function returns a reference to `Sq`.

Result type : **Sequence&**

`Sq.ImpConc(Sq1)`
Concatenates `Sq` and `Sq1` and places the result in `Sq`. The function returns a reference to `Sq`.

Result type : **Sequence&**

`Sq.Elems()`
Constructs a `Set` containing all elements in `Sq`. The `Set` is returned.

Result type : **Set**

`Sq.First(G)`
Returns 1 if `Sq` is not empty and returns the first element in the reference parameter `G`. If `Sq` is empty it returns 0 and G will be returned as an empty Generic.

Result type : **bool**

`Sq.Next(G)`
Returns 1 and the next element in the reference parameter `G`. If there are no more keys in `Sq` it returns 0 and G will be returned as an empty Generic.

Result type : **bool**

Examples

```
Sequence Sq1,Sq2;  Set St;  Int I(5);  Generic G;

if (Sq1.IsEmpty())
  wcout << "Sq1 is initialized to the empty sequence" << endl;
  wcout << Sq1.ImpAppend(I).ImpPrepend(St).Length()
     << " is the length of Sq1" << endl;
  wcout << Sq1.ImpTl().ascii()
     << " is the value of Sq1 after applying ImpTl" << endl;
Sq2.ImpAppend(Sq1.Hd());
  wcout << Sq1.ImpConc(Sq2).Length()
     << " is the length of Sq1 after applying ImpConc" << endl;
```

The result of running this program is :

```
Sq1 is initialized to the empty sequence
2 is the length of Sq1
[ 5 ] is the value of Sq1 after applying ImpTl
2 is the length of Sq1 after applying ImpConc
```

5.11 Set

`Set` supports the following member functions:

`Set St`
: Declares `St` as a value of type `Set` and initializes it to the empty set.

  Result type : **void**

`Set St(St1)`
: Declares `St` as a value of type `Set` which is equal to `St1`.

  Result type : **void**

`St.Insert(A)`
: Inserts `A` in `St`. If `A` already exists in `St`, `St` is left unchanged. The function returns a reference to `St`.

  Result type : **Set&**

`St.Card()`
: Returns an integer denoting the number of elements in `St`.

  Result type : **int**

`St.IsEmpty()`
: Returns `true` if `St.Card() == 0` and `false` otherwise.

  Result type : **bool**

`St.InSet(A)`
: Returns 1 if `A` is in `St` and 0 otherwise.

  Result type : **bool**

`St.ImpUnion(St1)`
: Adds all elements of `St1` to `St`. The function returns a reference to `St`.

  Result type : **Set&**

`St.ImpIntersect(St1)`
: Removes all elements of `St` not occuring in `St1`. The function returns a reference to `St`.

  Result type : **Set&**

`St.GetElem()`
: Returns an element `G` from `St`. If `St` is empty an error is signaled cf. Section 7.

  Result type : **Generic&**

`St.RemElem(A)`
: Removes `A` from `St`. If `A` is not in `St` an error is signaled cf. Section 7. The function returns a reference to `St`.

  Result type : **Set&**

`St.SubSet(St1)`
: Returns 1 if `St` is a subset of `St1` and 0 otherwise.

  Result type : **bool**

`St.ImpDiff(St1)`
: Removes all elements of `St1` from `St`. The function returns a reference to `St`.

  Result type : **Set&**

`St.First(G)`
: Returns 1 if `St` is not empty and returns the first element in the reference parameter `G`. If `St` is empty it returns 0 and G will be returned as an empty Generic.

  Result type : **bool**

`St.Next(G)`
: Returns 1 and the next element in the reference parameter `G`. If `St` is empty it returns 0 and G will be returned as an empty Generic.

  Result type : **bool**

Examples

```
Set St1,St2;  Int I(5);  Generic G;

St1.Insert(I).Insert(St2);
if (St1.InSet(I))
  wcout << "St1 contains I" << endl;
St2.Insert(I).Insert(St1);
  wcout << St1.ImpUnion(St2).ascii()
     << " is the union of St1 and St2" << endl;
  wcout << St1.ImpIntersect(St2).ascii()
     << " is the intersection of St1 and St2" << endl;
```

The result of running this program is :

```
St1 contains I
{ 5, {  }, { 5, {  } } } is the union of St1 and St2
{ 5, { 5, {  } } } is the intersection of St1 and St2
```

5.12 Record

`Record` supports the following member functions:

`Record Rc`
: Declares `Rc` as a value of type `Record`. The number of fields is set to 0, and the tag is set to 0.

Result type : **void**

`Record Rc(i1,i2)`
Declares `Rc` as a value of type `Record` with the tag `i1` and `i2` fields. Note that the tag value -1 is reserved and must therefore not be used.

Result type : **void**

`Record Rc(Rc1)`
Declares `Rc` as a value of type `Record` which is equal to `Rc1`.

Result type : **void**

`Rc.SetField(i,A)`
Modifies the `i`'th field to `A`. If `i` is not within the defined number of fields for `Rc` an error is signaled cf. Section 7. The function returns a reference to `Rc`.

Result type : **Record&**

`Rc.GetField(i)`
Returns the contents `G` of the `i`'th field of `Rc`. If `i` is not within the range of the defined number of fields for `Rc` an error is signaled cf. Section 7.

Result type : **Generic&**

`Rc.GetTag()`
Returns the tag `i` of `Rc`.

Result type : **int**

`Rc.Is(i)`
Returns 1 if `i` equals the tag of `Rc` and 0 otherwise.

Result type : **bool**

`Rc.Length()`
Returns the number of fields declared for `Rc`.

Result type : **int**

Examples

```
#define Ex 1

Record Rc1(Ex,2);  Int I(5);  Set St;
```

```
Rc1.SetField(1,I).SetField(2,St);
if (Rc1.Is(Ex))
  wcout << Rc1.GetTag() << " is the tag of Rc1" << endl;
  wcout << Rc1.GetField(1).ascii()
     << " is the value of the first field of Rc1" << endl;
  wcout << Rc1.Length() << " is the number of fields in Rc1" << endl;
```

The result of running this program is :

```
1 is the tag of Rc1
5 is the value of the first field of Rc1
2 is the number of fields in Rc1
```

5.12.1 The Record Information Map

It is not legal to define a record with the same tag with different size in the VDM Library. The VDM Library provides an internal state in which it is possible to define relations between tag, size and string tag of a record. The default internal st ate can be accessed through the function *VDMGetDefaultRecInfoMap*. The state and its member functions is defined in class *VDMRecInfoMap* that supports the following public member functions:

`VDMRecInfoMap ri`
: Declares `ri` as a value of type `VDMRecInfoMap`.

  Result type : **void**

`NewTag(int tag, int size)`
: Declares a new tag and size.

  Result type : **void**

`AskDontCare(int tag, int size, int field)`

: Returns `true` if field number *field* of the record declared with tag *tag* is an abstract field select.

  Result type : **bool**

`AskDontCare(int tag, int field)`

Returns `true` if field number *field* of the record declared with tag *tag* is an abstract field select.

Result type : **bool**

```
SetDontCare(int tag, int size, int field)
```

Mark field number `field` as an abstract field select for the record with tag number *tag*.

Result type : **void**

```
SetDontCare(int tag, int field)
```

Mark field number `field` as an abstract field select for the record with tag number *tag*.

Result type : **void**

```
SetSymTag(int tag, int size, const wstring& symtag)
```

Set the symbolic tag (the string tag of the record) of tag number *tag* to *symtag*. This will relate *tag* to *symtab* in the *VDMRecInfoMap*. When using the *ascii* method for printing out record values and if a symbolic tag has been defined for a specific tag number, the symbolic tag string will be printed instead of the tag number *tag*.

Result type : **void**

```
SetSymTag(int tag, const wstring& symtag)
```

Set the symbolic tag (the string tag of the record) of tag number *tag* to *symtag*. This will relate *tag* to *symtab* in the *VDMRecInfoMap*. When using the *ascii* method for printing out record values and if a symbolic tag has been defined for a specific tag number, the symbolic tag string will be printed instead of the tag number *tag*.

Result type : **void**

```
SetPrintFunction(int tag, int size, vdm_pp_function_ptr f)
```

With this function it is possible to relate a pointer to a function *f* to tag number *tag*. The function *f* will be used when calling the *ascii* method on the record with the tag *tag*.

Result type : **void**

`SetPrintFunction(int tag, int size, vdm_pp_function_ptr f)`

With this function it is possible to relate a pointer to a function *f* to tag number *tag*. The function *f* will be used when calling the *ascii* method on the record with the tag *tag*.

Result type : **void**

`GetSize(int tag)`

Returns the size of the record that was declared with tag number *tag*.

Result type : **int**

`GetSymTag(int tag, wstring & s)`

Extracts the symbolic string tag and assigns it to wstring *s* of the record *tag*.

Result type : **bool**

`size()`

Returns the size of the map *VDMRecInfoMap*.

Result type : **int**

`dump(ostream & o`

Prints the information in the *VDMRecInfoMap* to the ostream *o*.

Result type : **void**

5.13 Tuple

`Tuple` supports the following member functions:

`Tuple Tp`

Declares `Tp` as a value of type `Tuple`. The number of fields is set to 0.

Result type : **void**

`Tuple Tp(i)`
: Declares `Tp` as a value of type `Tuple` with `i` fields.

  Result type : **void**

`Tuple Tp(Tp1)`
: Declares `Tp` as a value of type `Tuple` which is equal to `Tp1`.

  Result type : **void**

`Tp.SetField(i,A)`
: Modifies the `i`'th field to `A`. If `i` is not within the defined number of fields for `Tp` an error is signaled cf. Section 7. The function returns a reference to `Tp`.

  Result type : **Tuple&**

`Tp.GetField(i)`
: Returns the contents `G` of the `i`'th field of `Tp`. If `i` is not within the defined number of fields for `Tp` an error is signaled cf. Section 7.

  Result type : **Generic&**

`Tp.Length()`
: Returns the number of fields declared for `Tp`.

  Result type : **int**

Examples

```
Tuple Tp1(2);  Int I(5);  Set St;

Tp1.SetField(1,I).SetField(2,St);
  wcout << Tp1.GetField(1).ascii()
     << " is the value of the first field of Tp1" << endl;
  wcout << Tp1.Length() << " is the number of fields in Tp1" << endl;
```

The result of running this program is :

```
5 is the value of the first field of Tp1
2 is the number of fields in Rc1
```

5.14 ObjectRef

The class ObjectRef is used to implement the object reference type in VDM^{++} [2]. This class should only be used to contain references to instances of classes generated by the *VDM++ to C++ Code Generator* [1].

ObjectRef contains a reference (pointer) to an instance of a C++ class, and a type variable identifying the type of the instance pointed to by the refernce pointer.

As the implementation of the VDM C++ library is based on reference counters, it will delete the pointer to the instance of a class when no existing objects of class `ObjectRef` refer to it.

`ObjectRef` supports the following member functions:

`ObjectRef Ob (p)`
: Declares `Ob` as a value of type `ObjectRef`. The reference pointer is set to `p`. A reference pointer must have been created with the C++ `new` operator, in order for the built-in garbage collector to work correctly. The `NULL` pointer will be used as default parameter, if no pointer is specified.

  Result type : **void**

`ObjectRef Ob(Ob1)`
: Declares `Ob` as a value of type `ObjectRef` which is equal to `Ob1`.

  Result type : **void**

`Ob.MyObjectId()`
: Returns the type `i` of `Ob`. The type is an integer defined as an enumeration type in `CGBase.h`

  Result type : **int**

`Ob.GetRef()`
: Returns the vdmBase pointer `p` of `Ob`.

  Result type : **vdmBase***

`Ob.SameBaseClass(Ob1)`
: Returns `True` if `Ob` and `Ob1` has same base class, i.e. if `Ob` and `Ob1` are instances of classes that can be derived from the same root superclass, and `False` otherwise.

  Result type : **Bool**

`Ob.IsOfClass(i)`
: Returns `True` if `Ob` refers to an object of a class with type `i` or any subclasses of `i`, and `False` otherwise. `i` is the type returned by `MyObjectId`.

  Result type : **Bool**

`Ob.IsOfBaseClass(i)`
: Returns `True` if the class with type `i` is a root superclass in the inheritance chain of the object referenced to by `Ob`, and `False` otherwise.

  Result type : **Bool**

`Ob.ascii()`
: The result of the `ascii()` method for an `ObjectRef` returns the type field and the reference pointer (hex format).

5.14.1 CGBase

The ObjectRef class is related to the code generated class CGBase. This class is superclass for the non-derived VDM++ super classes. Together with this class is also defined a function casting from an object reference to a pointer to a code generated class and and enumeration type that is used to uniquely tagging of each code generated VDM++ class. This is defined in the header file `CGBase.h` and implemented in `CGBase.cc`.

`ObjGet_`***class-name***`(Ob)`
: Returns the pointer to the code generated class *class-name*. If `Ob` does not refer to *class-name* zero is returned.

  Result type : ***class-name ****

Examples

Consider the following VDM^{++} class:

```
class A

operations
  Test: () ==> nat
  Test()  ==
```

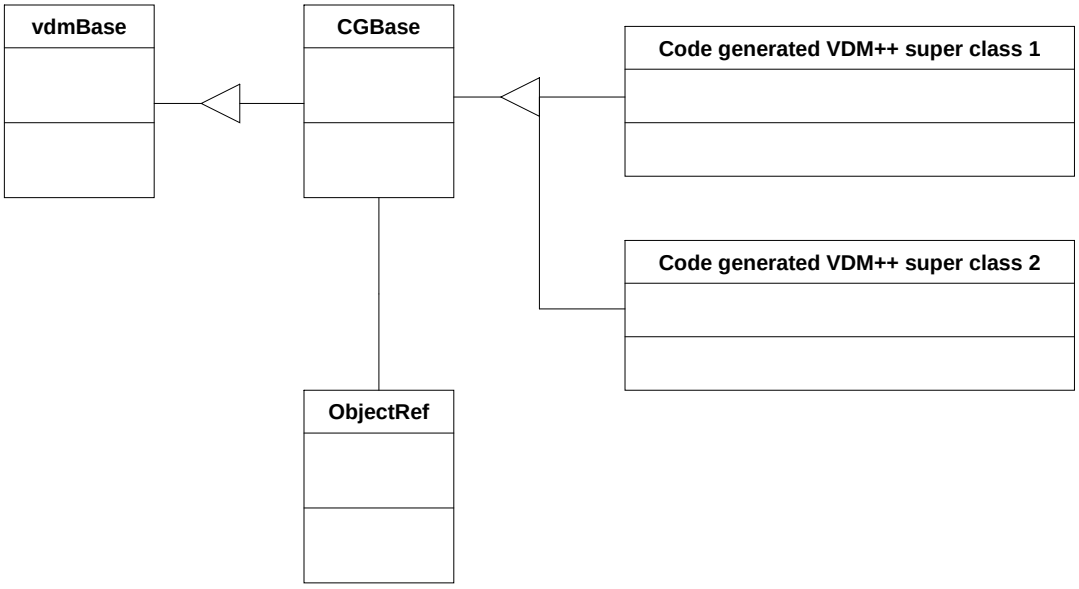

Figure 3: Relation between C++ classes

```
    let a = 10 + 10 in
      return a

end A
```

The following header file, `A.h`, is generated by the *VDM++ to C++ Code Generator*:

```
#ifndef _A_h
#define _A_h

#include <math.h>
#include "metaiv.h"
#include "cg.h"
#include "cg_aux.h"
#include "CGBase.h"

class vdm_A : public virtual CGBase {
public:
  virtual Int vdm_Test();
```

```
  vdm_A();
  virtual ~vdm_A() {}
};

#endif
```

You can now use ObjectRef to implement object references of type `A` in the following way:

```
#include <iostream.h>
#include "A.h"

int main ()
{
  Sequence sq; Int i (10);
  ObjectRef cls1 (new vdm_A());
  ObjectRef cls2 (new vdm_A());

  sq.ImpAppend (i).ImpAppend (cls1).ImpAppend (cls2);

  wcout << VDM_A << " is the value of VDM_A" << endl;
  wcout << sq.ascii () << " is the value of sq" << endl;

  ObjectRef cls3 (sq[2]);
  if (cls3.MyObjectId () == VDM_A) {
    vdm_A* cp = ObjGet_vdm_A(cls3);
    wcout << cp->vdm_Test ().ascii () << " is the result of Test ()" << endl;
    wcout << cls1 == cls2 << " is the result of cls1 == cls2" << endl;
    wcout << cls1 == cls3 << " is the result of cls1 == cls3" << endl;
  }
  else
    wcout << "Something strange happened!" << endl;
}
```

The result of running this program is :

```
1 is the value of VDM_A
[ 10, @(1, 373776), @(1, 380808) ] is the value of sq
20 is the result of Test ()
0 is the result of cls1 == cls2
1 is the result of cls1 == cls3
```

5.15 Generic

`Generic` supports the following member functions:

`Generic G`
: Declares `G` as a value of type `Generic`. The value is an instance of GenericVal.

  Result type : **void**

`Generic G(A)`
: Declares `G` as a value of type `Generic` which value is equal to the value of `A`.

  Result type : **void**

6 SETs, SEQuences and MAPs

For the types set, sequence and maps corresponding C++ templates exists. These templates make it possible declare types with better type information. Using these types it is possible to declare not only a set but also which kind of value type the set can contain.

6.1 SETs

The `SET` template is derived from the `Set` class.

The `SET` template support the constructor functions:

`SET<A> St`
: Declares `St` as a `Set` of type `A`. The value of `St` is initialised to the empty set.

  Result type: **void**

`SET<A> St(St1)`
: Declares `St` as a `Set` of type `A`. The value of `St` is initialised to the value of `St1`.

  Result type: **void**

In addition the same functions and operators that work on the `Set` class are also declared for the `SET` template.

6.2 SEQuences

The `SEQ` template is derived from the `Sequence` class.

The `SEQ` template support the constructor functions:

`SEQ<A> Sq`
: Declares `Sq` as a `Sequence` of type `A`. The value of `Sq` is initialised to the empty sequence.

  Result type: **void**

`SEQ<A> Sq(Sq1)`
: Declares `Sq` as a `Sequence` of type `A`. The value of `Sq` is initialised to the value of `Sq1`.

  Result type: **void**

In addition the same functions and operators that work on the `Sequence` class are also declared for the `SEQ` template.

6.3 MAPs

The `MAP` template is derived from the `Map` class.

The `MAP` template supports the constructor functions:

`MAP<A,B> M`
: Declares `M` as a `Map` from type `A` to `B`. The value of `M` is initialised to the empty map.

  Result type: **void**

`MAP<A, B> M(M1)`
: Declares `M` as a `Map` from type `A` to `B`. The value of `M` is initialised to the value of `M1`.

  Result type: **void**

In addition the same functions and operators that work on the `Map` class are also declared for the `MAP` template.

7 Error messages

When an error is detected by the library functions, an error number along with a string describing the library function which detected the error will be written to the m4err stream. Then the 'exit' function is called to terminate the program.

The errors have been divided into two categories. User errors (U) are errors that can appear under normal use of the library. An example could be trying to extract an element from an empty set.

Internal errors (I) are more severe errors. These errors should not appear under normal use of the libraries.

No.	*Symbolic name*	*Description*	*Error type*
1	ML_CONFLICTING_RNGVAL	Insert a key in a map which already exists with a different range value	U
2	ML_NOT_IN_DOM	Applying a map with a key which is not in domain	U
3	ML_CAST_ERROR	Generic casted to wrong type	U
4	ML_INDEX_OUT_OF_RANGE	Index out of range in sequence, tuple or record function	U
5	ML_OP_ON_EMPTY_SEQ	Illegal function on empty sequence	U
6	ML_OP_ON_EMPTY_SET	Illegal function on empty set	U
7	ML_NOT_IN_SET	Tried to remove non-existing element	U
8	ML_ASSIGN_ERROR	Tried to assign two variables of different types	U
9	ML_TRAVERSE_CONFLICT	Error detected while evaluating the member function next	I
10	ML_HD_ON_EMPTY_SEQUENCE	Tried to take hd on an empty sequence	U
11	ML_TL_ON_EMPTY_SEQUENCE	Tried to take tl on an empty sequence	U
12	ML_RANGE_ERROR	Index out of range for tuple or record	U
13	ML_ZERO_REFCOUNT	Zero refcount detected	I
14	ML_NULL_REF	Zero pointer reference	I
15	ML_DIV_BY_ZERO	Division by zero.	U

References

[1] CSK. *The VDM++ to C++ Code Generator*. CSK.

[2] Dürr, E., and (editor), N. P. VDM++ Language Reference Manual. Afrodite (esprit-iii project number 6500) document, Cap Volmac, August 1995.
Afrodite Doc.id : AFRO/CG/ED/LRM/V11.

A Files

The interface to the VDM C++ Library consists of the following files:

metaiv.h is the header file containing the prototypes of all the type specific functions described in section 5.

libvdm.a is a library archive containing the implementation of all the functions described in this document.