九州大学
KYUSHU UNIVERSITY

VDMTools

VDM C++ ライブラリ

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

VDM C++ ライブラリ 1.0

— Revised for VDMTools v9.0.6

目 次

1 導入

本書では、VDM C++ ライブラリを構成するクラスとメソッドの記述を行う。本書を読むためには、いくらかの C++ の知識が必要となる。各々 VDM 型に対して、その型を実装する相当する C++ クラスが 1 つ存在する。加えて合成型に対しては、VDM C++ ライブラリには集合、写像、列のテンプレートがあり、更に良質な型情報を含む型宣言を可能にしている。

第 2 章で、本書で使用する表記法が一覧されている。第 3 章では、VDM オブジェクトの一般的構造を簡単に示す。第 4 章ですべての VDM クラスに共通する関数を記述し、他方では第 5 章で各々 VDM クラスにおいて実行可能な特有関数が一覧されている。集合、写像、列のためのテンプレートは、第 6 章に記されている。第 7 章には、全エラーメッセージを記述する。

付録 A には、VDM C++ ライブラリを構成するファイルが一覧されている。

ライブラリの最新版は次のプラットフォーム上で利用可能である:

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

2 表記法

本書では以下の慣例を用いる:

変数名	変数型	*C++* クラス
i	C++ 整数型	int
c	C++ 文字型	char
d	C++ ダブル型	double
s	C++ 文字列型	string

変数名	変数型	C++ クラス
I	VDM 整数型	Int
M	VDM 写像型	Map
C	VDM 文字型	Char
B	VDM ブール型	Bool
N	VDM Nil 型	Nil
Q	VDM 引用型	Quote
G	VDM 汎用型	Generic
Rl	VDM 実数型	Real
Rc	VDM レコード型	Record
Tx	VDM テキスト型	Text
Tp	VDM 組型	Tuple
Tk	VDM トークン型	Token
St	VDM 集合型	Set
Sq	VDM 列型	Sequence
Ob	VDM オブジェクト参照型	ObjectRef
A	上記 VDM 型のいずれか	

3 VDM 値の一般的構造

この章では、“水面下” でどのようなことが行われているかの考え方を示すため、VDM 値の一般的構造を簡単に述べていく。VDM C++ クラスを用いるには、このようなある程度の基本的知識をもつことが重要である。

Set、Int、Map 等の VDM クラスすべてに対して、SetVal、IntVal、MapVal 等という名称をもつ相当する値クラスが 1 つずつ存在する。すべての VDM クラスは Common クラスのサブクラスで、一方すべての値クラスは MetaivVal クラスのサブクラスである。

たとえば変数 I が Int 型で宣言されているならば、Int 型と IntVal 型のインスタンスが生成され Int インスタンスからの ポインタ p (このポインタは実際には Common で定義される) が IntVal インスタンス (実際には IntVal インスタンスの MetaivVal 部分) を指し示している。この状態は図 1 で描かれている。破線は

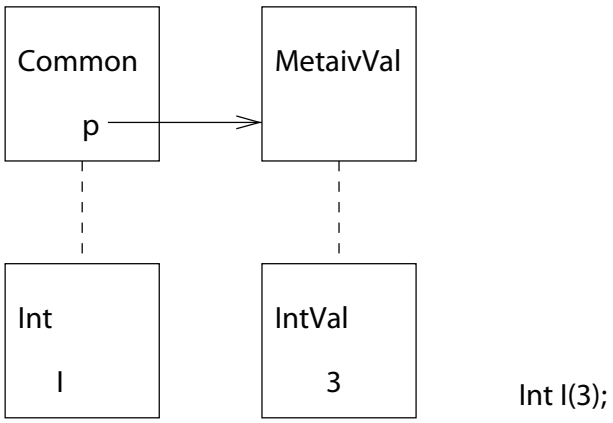

図 1: 一般的構造

クラス階層を表し、実線はポインタ p を示す。I の値は IntVal インスタンス中に置かれている。以降で、変数値の参照とは常に、ポインタ p が指し示す Val クラスのインスタンスを意味する。

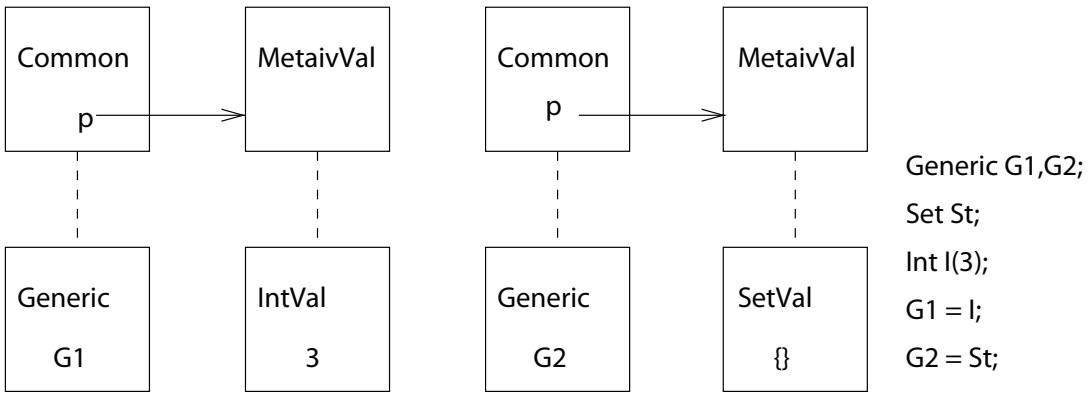

図 2: 汎用クラス

合併型の考え方をサポートするものとして、 汎用クラスが導入されている。これは合成 VDM 型 (写像、列、組、集合、レコード) を示すクラスのインスタンスが、同時に異なる型の要素を含めることを許す。汎用値としては、基本の IntVal,RealVal から TupleVal, SequenceVal といった合成値まで任意の VDM 値であってよい。このことは、 汎用型の変数は任意の VDM 型の基礎となる値を含めることが可能であることを意味する。図 2 では汎用の例題を 2 つ示す。

現実には、合成型は汎用型の要素だけを含めることができるように実装されてい

る。合成型に要素を挿入する関数のほとんどは、挿入前に要素を自動的に 汎用型にキャストすることができるが、要素を取り出す関数は常に 汎用型を返す。

この例を示すために VDM 式を見てみよう:

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

列 Sq は整数と引用の両方を含む。C++ が強く型付けされているので、 let-式の C++ 実装では、 Generic に対する整数と引用のキャストを、列 Sq に追加する前に行わなければならない。以下の実装において、このキャストは ImpAppend 関数により自動的になされ、取り出し関数である Hd() は Generic を返す。この列の最初の要素は Int 型なので、その後 G は Int にキャストすることができる。

```
Sequence Sq;  Generic G;

Sq.ImpAppend(Int(1));
Sq.ImpAppend(Quote(‘‘<TWO>’’));

G = Sq.Hd();
```

一般的にはどのような型も Generic にキャストされることは可能であるが、汎用であるのは表面上のみであり、基礎となる値は保持されている。たとえば、図 2 上の汎用 G1 は後から逆のキャストで Int にもどすことが可能である。合成型 (写像、列、組、集合、レコード) を表わすすべてのクラスに対して、そのクラスに含まれるすべての要素が含まれる前に自動的に Generic にキャストされる。これは、合成変数から要素が取り出されるときは常に Generic 型になることを意味する。必要ならば、元の型へ逆キャストを行うことも可能である。

4 VDM 型の一般関数

この章では、 Common クラスで定義されているすべての型のインスタンスに適用可能な全ての関数を記述する。[1]

[1] VDMTools 内部の文字コードは Unicode を使用しているため、VDM 型と wstring を交換する場合、文字コードに注意する。

関数

`A.MyValType()`

`A` の値の型を返す。

結果型：**metaivType**

列挙型 `metaivType` がライブラリの一部であることに注意する。

```
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`

`A1` に `A2` の値を与える。`A1` が `Generic` 型であるならば、 `A2` は任意の型の可能性がある。この場合、 `A1` はそのまま汎用型であるが `A2` の値を含むことになる。それ以外の場合は、 `A2` の値の型は `A1` の値の型と同じでなければならない。

結果型：A1 への参照

`A1 == A2`

`A1` の値が `A2` の値と等しければ `true` を返し、それ以外は `false` を返す。

結果型：**bool**

`A1 != A2`

`A1` の値が `A2` の値と等しくなければ `true` を返し、それ以外は `false` を返す。

結果型：**bool**

`A.ascii()`

VDM 値の ASCII 表現を含む 文字列 を返す。

結果型：**wstring**

A.IsNil()
A の値が Nil 型であるならば true を返す
結果型：bool

A.IsChar()
A が Char 型であるならば true を返す
結果型：bool

A.IsInt()
A が Int 型であるならば true を返す
結果型：bool

A.IsReal()
A が Real 型であるならば true を返す
結果型：bool

A.IsQuote()
A が Quote 型であるならば true を返す
結果型：bool

A.IsTuple()
A が Tuple 型であるならば true を返す
結果型：bool

A.IsRecord()
A が Record 型であるならば true を返す
結果型：bool

A.IsSet()
A が Set 型であるならば true を返す
結果型：bool

A.IsMap()
A が Map 型であるならば true を返す
結果型：bool

`A.IsText()`
: `A` が `Text` 型であるならば `true` を返す

  結果型 : **bool**

`A.IsToken()`
: `A` が `Token` 型であるならば `true` を返す

  結果型 : **bool**

`A.IsBool()`
: `A` が `Bool` 型であるならば `true` を返す

  結果型 : **bool**

`A.IsSequence()`
: `A` が `Sequence` 型であるならば `true` を返す

  結果型 : **bool**

`A.IsObjectRef()`
: `A` が `ObjectRef` 型であるならば `true` を返す

  結果型 : **bool**

`A.WriteVal(o)`
: `A` の値を、関数 ReadVal を用いて再び値を読み込むことができる形式で `ostream o` に書き込む。`ReadVal` と `WriteVal` は、ファイルシステムに値を保存するために用いられ、後で読み戻しを行う (永続性)。

  結果型: **void**

`Generic g; g = ReadVal(i)`
: (istream を通して) WriteVal メソッドで書かれたファイルから 値を読み込む。`ReadVal` は関数であり、メソッドではないことに注意する。

  `WriteVal` が書き込み `ReadVal` が読み込む形式は低レベルフォーマットであって、人が取り扱うためのものではない。

  結果型: **Generic**

例題

この節で述べられた関数を用いた例題 C++ プログラム。

```
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;
```

このプログラムの実行結果は次の通り：

```
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 出力ワイドストリームへ値の印刷

`os << v` と表すことで任意の VDM 値 `v` を 出力ワイドストリーム `os` へ印刷することが可能である。

レコードは、既定では `mk_unknown4(...)` というように、数値タグと共に印刷されることになる。また、タグ名を文字形式で印刷するには、5.12.1 The Record Information Map を使用する。

例題

```
#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;
}
```

このプログラムの実行結果は次の通り：

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

5 VDM 型の特有関数

5.1 Int

Int は以下の要素関数をサポートしている:

Int I
: I を Int として宣言する。I の値は 0 に初期化される。

結果型：**void**

Int I(i)
: I を Int として宣言し、初期値を i とする。

結果型：**void**

`Int I(I1)`
: `I` を `Int` として宣言し、値は `I1` と等しいとする。

結果型：**void**

`I.GetValue()`
: `I` の C++ 整数値を返す。

結果型：**int**

`I = i`
: `i` の値を `I` に与える。

結果型：**Int&**

`-I`
: 単項マイナス。`I` の正負逆符号の値をもつ `Int` を返す。

結果型：**Int**

`I1 + I2`
: 2 項加算。

結果型：**Int**

`I + R`
: 2 項加算。

結果型：**Real**

`I1 - I2`
: 2 項減算。

結果型：**Int**

`I - R`
: 2 項減算。

結果型：**Real**

`I1 * I2`
: 2 項乗算。

結果型：**Int**

`I * R`
　２項乗算。
　結果型：**Real**

`I1 / I2`
　２項除算。
　ゼロによる除算はエラー 第 7 章参照 を起こす。
　結果型：**Real**

`I / R`
　２項除算。
　ゼロによる除算はエラー 第 7 章参照 を起こす。
　結果型：**Real**

`I1.Exp(I2)`
　指数演算。
　結果型：**Real**

`I.Exp(R)`
　指数演算。
　結果型：**Real**

例題

```
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;
```

このプログラムの実行結果は以下の通り：

```
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 は以下の要素関数をサポートしている:

Real Rl
: Rl を Real として宣言する。Rl の値は 0 に初期化される。

結果型 : **void**

Real Rl(d)
: Rl を Real として宣言し、値に d で初期化する。

結果型 : **void**

Real Rl(Rl1)
: Rl を Real として宣言し、値は Rl1 に等しい。

結果型 : **void**

Rl.GetValue()
: Rl の C++ double 値を返す。

結果型 : **double**

Rl = d
: Rl に d の値を与える。

結果型 : **Real&**

-R
: 単項マイナス。 R の逆符号の値をもつ Real を返す。

結果型 : **Real**

R1 + R2
: 2 項加算。

結果型 : **Real**

`R + I`
　　2 項加算。
　　結果型：**Real**

`R1 - R2`
　　2 項減算。
　　結果型：**Real**

`R - I`
　　2 項減算。
　　結果型：**Real**

`R1 * R2`
　　2 項乗算。
　　結果型：**Real**

`R * I`
　　2 項乗算。
　　結果型：**Real**

`R1 / R2`
　　2 項除算。
　　結果型：**Real**

`R / I`
　　2 項除算。
　　結果型：**Real**

`R1.Exp(R2)`
　　指数演算。
　　結果型：**Real**

`R.Exp(I)`
　　指数演算。
　　結果型：**Real**

例題

```
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;
```

このプログラムの実行結果は次の通り :

```
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 は以下の要素関数をサポートする:

Bool B
: B を Bool として宣言する。 B の値は 0 (false) に初期化される。

結果型 : **void**

Bool B(i)
: B を Bool として宣言し、値は i に初期化する。

結果型 : **void**

Bool B(B1)
: B を Bool として宣言し、値は B1 に等しい。

結果型 : **void**

`B.GetValue()`
: B の C++ **bool** 値を返す。

結果型：**bool**

`B = i`
: i の値を B に与える。

結果型：**Bool&**

`B.mnot()`
: 論理否定。

結果型：**Bool**

`!B`
: 論理否定。

結果型：**Bool**

`B1.mand(B2)`
: 論理積。

結果型：**Bool**

`B1 && B2`
: 論理積。

結果型：**Bool**

`B1.mor(B2)`
: 論理和。

結果型：**Bool**

`B1 || B2`
: 論理和。

結果型：**Bool**

例題

```
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 = true;
  wcout << B2.GetValue() << "  is the new value of B2 (true)" << endl;
```

この実行プログラムの結果は次の通り :

```
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 は以下の要素関数をサポートする:

Nil N
: N を Nil 型の値として宣言する。
結果型 : **void**

Nil 型のインスタンスは全型に共通の関数のみをサポートする (第 4 章を参照)。

5.5 Quote

Quote は以下の要素関数をサポートする:

Quote Q
: Q を Quote 型の値として宣言する。 Q の値は ""に初期化される。
結果型 : **void**

`Quote Q(s)`

`Q` を `Quote` 型の値として宣言し、値を `s` に初期化する。

結果型：**void**

`Quote Q(Q1)`

`Q` を、`Q1` と等しい `Quote` 型の 値として宣言する。

結果型：**void**

`Q.GetValue()`

`Q` の C++ ワイド文字列値を返す。

結果型：**wstring**

`Q = s`

`s` の値を `Q` に与える。

結果型：**Quote&**

例題

```
Quote Q1(L"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 = L"Q_TWO";
  wcout << Q2.GetValue() << "  is the new value of Q2" << endl;
```

このプログラムの実行結果は次の通り：

```
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 は以下の要素関数をサポートする:

Char C
: C を Char 型の値として宣言する。 C の値は '?' に初期化される。

結果型 : **void**

Char C(c)
: C を Char 型の値として宣言し、値を c と初期化する。

結果型 : **void**

Char C(C1)
: C を、C1 と等しい Char 型の値として宣言する。

結果型 : **void**

C.GetValue()
: C の C++ ワイド文字値を返す。

結果型 : **wchar_t**

C = c
: C に c の値を与える。

結果型 : **Char&**

例題

```
Char C1(L'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 = L'd';
  wcout << C2.GetValue() << "  is the new value of C2" << endl;
```

このプログラムの実行結果は次の通り：

```
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 は以下の要素関数をサポートする:

Text Tx
: Tx を Text 型の値として宣言する。 Tx の値は ""に初期化される。

結果型：**void**

Text Tx(s)
: Tx を Text 型 の値として宣言し、値を s に初期化する。

結果型：**void**

Text Tx(Tx1)
: Tx を、Tx1 と等しい値を持つ Text 型 の値として宣言する。

結果型：**void**

Tx.GetValue()
: Tx の C++ ワイド文字列値を返す。

結果型：**wstring**

Tx = s
: Tx に s の値を与える。

結果型：**Text&**

例題

```
Text Tx1(L"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 = L"Tx_TWO";
  wcout << Tx2.GetValue() << "  is the new value of Tx2" << endl;
```

このプログラムの実行結果は次の通り：

```
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 は以下の要素関数をサポートする:

Token Tk
: Tk を Token 型の値として宣言する。 Tk の値は ""に初期化されている。

結果型：**void**

Token Tk(s)
: Tk を Token 型の値として宣言し、値を s に初期化する。

結果型：**void**

Token Tk(Tk1)
: Tk を Tk1 と等しい Token 型 の値として宣言する。

結果型：**void**

`Tk.GetValue()`
`Tk` の C++ ワイド文字列値を返す。
結果型：**wstring**

`Tk = s`
`Tk` に `s` の値を与える。
結果型：**Token&**

例題

```
Token Tk1(L"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 = L"Tk_TWO";
  wcout << Tk2.GetValue() << "  is the new value of Tk2" << endl;
```

このプログラムの実行結果は次の通り：

```
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` は以下の要素関数をサポートする:

`Map M`
`M` を `Map` 型の値として宣言し、空写像に初期化する。
結果型：**void**

`Map M(M1)`

M を Map 型の値として宣言し、 M1 に初期化する。

結果型：**void**

`M.Insert(A1, A2)`

キー A1 をそれに伴う内容 A2 と共に M に挿入する。キー A1 がすでに M の定義域に属している場合は、 A2 が値域値と等しいかどうかがチェックされる。等しくなければ、エラー 第 7 章参照 が起こされる。関数は M への参照を返す。

結果型：**Map&**

`M.ImpModify(A1, A2)`

キー A1 が M の定義域に既に属しているならば値域値が A2 に変更されるということを除けば、 Insert として作用する。関数は M への参照を返す。

結果型：**Map&**

`M[A]`

キー A と関連した内容を返す。A が M に対する定義域にない場合はエラー 第 7 章参照 が起こされる。

結果型：**Generic&**

`M.ImpOverride(M1)`

M は M と M1 の和になる。M に存在するキーが M1 にも存在するときは、M の相当する内容は M1 の内容で上書される。関数は M への参照を返す。

結果型：**Map&**

`M.Size()`

M のキー数を意味する整数を返す。

結果型：**int**

`M.IsEmpty()`

`M.Size() == 0` ならば `true` を、他の場合は `false` を返す。

結果型：**bool**

`M.Dom()`

M のすべてのキーを含む集合を返す。

結果型：**Set**

M.Rng()
M のすべての内容を含む集合を返す。
結果型：**Set**

M.DomExists(A)
A が M の定義域に含まれる場合は true をそうでない場合は false を返す。
結果型：**bool**

M.RemElem(A)
A が M の定義域中にあるならキーと内容の両方を削除し、そうでない場合はエラー 第 7 章参照 を起こす。
結果型：**Map&**

M.First(G)
M が空でないならば true を返し、内部的な順にしたがって、参照パラメータ G に最初のキーを返す。M が空であるならば false を返し、汎用型の空として G が返される。
結果型：**bool**

M.Next(G)
次のキーが存在すれば true および参照パラメータ G に次のキー を返す。M にそれ以上のキーが存在しない場合 false を返し、汎用型の空として G が返される。
結果型：**bool**

例題

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

このプログラムの実行結果は次の通り：

```
{ 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

Sequence の要素は 1 から列長まで添え字付けされる。Sequence は以下の要素関数をサポートする:

`Sequence Sq`

Sq を Sequence 型の値として宣言し、空列に初期化する。

結果型：**void**

`Sequence Sq(Sq1)`

Sq を Sequence 型の値として宣言し、 Sq1 に初期化する。

結果型：**void**

`Sequence Sq(s)`

Sequence 型の値として Sq を宣言し、“seq of char” に変換された wstring s に初期化する。

結果型：**void**

`Sq[i]`

Sq の i 番目の要素を返す。i が有効な添え字でないならば、エラー 第 ¯ 7 章参照 が起こされる。

結果型： **Generic&**

`Sq.Index(i)`

Sq の i 番目の要素を返す。i が有効な添え字でないならば、エラー 第 ¯ 7 章参照 が起こされる。

結果型： **Generic&**

`Sq.Hd()`

Sq の先頭の要素が返される。 Sq が空列ならば、エラー 第 ¯ 7 章参照 が起こされる。

結果型： **Generic&**

`Sq.Tl()`

Sq の尾部を返す。Sq が空列であるならば、エラー 第 ¯ 7 章参照 が起こされる。

結果型： **Sequence**

`Sq.ImpTl()`

Sq を Sq の尾部に置き換える。Sq が空列であるならば、エラー 第 ¯ 7 章参照 が起こされる。

関数は Sq への参照を返す。

結果型： **Sequence&**

`Sq.RemElem(int i)`

i が Sq に対して有効な添え字であれば Sq から i 番目の要素を削除する。そうでなければ、エラー 第 ¯ 7 章参照 が起こされる。

関数は Sq への参照を返す。

結果型: **Sequence&**

`Sq.Length()`

Sq の要素数を意味する整数を返す。

結果型： **int**

Sq.GetString(wstring& str)

“seq of char” を wstring に変換する。

Sq の全要素が文字型であるならば true を返し、パラメータ str 中の Sq の文字列表現に設定する。そうでない場合は GetString が false を返し、str を空の文字列 ("") に設定する。

結果型：**bool**

Sq.IsEmpty()

Sq.Size() == 0 ならば true を、他の場合は false を返す。

結果型：**bool**

Sq.ImpAppend(A)

A を Sq の後に付けてその結果を Sq に入れる。関数は Sq への参照を返す。

結果型：**Sequence&**

Sq.ImpModify(i,A)

Sq の i 番目の要素を A に修正する。i が有効な添え字でないならば、エラー 第 ¯ 7 章参照 が起こされる。関数は Sq への参照を返す。

結果型：**Sequence&**

Sq.ImpPrepend(A)

Sq の先頭に A を追加して、その結果を Sq に入れる。関数は Sq への参照を返す。

結果型：**Sequence&**

Sq.ImpConc(Sq1)

Sq と Sq1 を連結して結果を Sq に入れる。関数は Sq への参照を返す。

結果型：**Sequence&**

Sq.Elems()

Sq のすべての要素を含む Set を構成する。 Set が返される。

結果型：**Set**

Sq.First(G)

Sq が空でないならば true を返し、参照パラメータ G に最初の要素を返す。Sq が空ならば false を返し、汎用型の空として G が返される。

結果型：**bool**

Sq.Next(G)

次のキーが存在すれば、true および参照パラメータ G の次の要素を返す。Sq にはさらなるキーがない場合には false が返され、 汎用型の空として G が返される。

結果型 : **bool**

例題

```
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;
```

このプログラムの実行結果は次の通り :

```
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 は以下の要素関数をサポートする:

`Set St`
: `Set` 型の値として `St` を宣言し、空集合に初期化する。

結果型：**void**

`Set St(St1)`
: `St` を、`St1` と等しい `Set` 型の値として宣言する。

結果型：**void**

`St.Insert(A)`
: `St` に `A` を挿入する。`A` が `St` に既に存在する場合は、 `St` は変更されない。関数は `St` への参照を返す。

結果型：**Set&**

`St.Card()`
: `St` の要素数を意味する整数を返す。

結果型：**int**

`St.IsEmpty()`
: `St.Card() == 0` ならば `true` を返しそうでない場合は `false` を返す。

結果型：**bool**

`St.InSet(A)`
: `A` が `St` にあるならば `true` を返し、その他の場合は `false` を返す。

結果型：**bool**

`St.ImpUnion(St1)`
: `St1` のすべての要素を `St` に加える。関数は `St` への参照を返す。

結果型：**Set&**

`St.ImpIntersect(St1)`
: `St1` に現れない `St` の全要素を削除する。関数は `St` への参照を返す。

結果型：**Set&**

`St.GetElem()`
: `St` から要素 `G` を返す。`St` が空ならば、エラー 第 7 章参照 が起こされる。

結果型：**Generic&**

St.RemElem(A)

St から A を取り除く。A が St に含まれない場合はエラー 第 7 章参照 が起こされる。関数は St への参照を返す。

結果型：**Set&**

St.SubSet(St1)

St が St1 の部分集合ならば true を返し、それ以外の場合は false を返す。

結果型：**bool**

St.ImpDiff(St1)

St から St1 の全要素を削除する。関数は St への参照を返す。

結果型：**Set&**

St.First(G)

St が空でないならば true を返し、参照パラメータ G 中に最初の要素を返す。St が空ならば false を返し、 G が汎用型の空として返される。

結果型：**bool**

St.Next(G)

次のキーが存在すれば、true および参照パラメータ G の次の要素を返す。St が空ならば false を返し、 G が汎用型の空として返される。

結果型：**bool**

例題

```
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;
```

このプログラムの実行結果は次の通り：

```
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 は以下の要素関数をサポートする:

Record Rc
: Rc を Record 型の値として宣言する。項目番号を 0、タグを 0、に設定する。.

  結果型：**void**

Record Rc(i1,i2)
: Rc を、タグ i1 と 項目 i2 をもった Record 型の値として宣言する。タグ値 -1 が留保され、したがって使用されてはならないことに注意する。

  結果型：**void**

Record Rc(Rc1)
: Rc を、Rc1 と等しい Record 型の値として宣言する。

  結果型：**void**

Rc.SetField(i,A)
: i 番目項目を A に修正する。Rc に対して定義された項目番号以内に i がない場合は、エラーが 第 ¯ 7 章参照 に起こされる。関数は Rc への参照を返す。

  結果型：**Record&**

Rc.GetField(i)
: Rc の i 番目の項目の内容 G を返す。i が Rc に対して定義された項目番号の範囲以内でない場合、エラー 第 ¯ 7 章参照 が起こされる。

  結果型：**Generic&**

Rc.GetTag()
Rc のタグ i を返す。
結果型：**int**

Rc.Is(i)
i が Rc のタグに等しいならば true を返し、そうでない場合は false を返す。
結果型：**bool**

Rc.Length()
Rc に対して宣言された項目数を返す。
結果型：**int**

例題

```
#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;
```

このプログラムの実行結果は次の通り：

```
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

VDM ライブラリ内での同じ名前で異なるサイズのレコードを定義することは、正当ではない。VDM ライブラリはレコードのタグ、サイズ、文字列タグ間の関係を定義することが可能な内部状態を提供する。既定の内部状態には、関数 *VDMGetDefaultRecInfoMap* を通してアクセス可能である。この状態とその要素関数は、クラス *VDMRecInfoMap* で定義されていて、以下のパブリックな要素関数をサポートしている:

`VDMRecInfoMap ri`
: `ri` を `VDMRecInfoMap` 型の値として宣言する。

結果型 : **void**

`NewTag(int tag, int size)`
: 新しいタグとサイズを宣言する。

結果型 : **void**

`AskDontCare(int tag, int size, int field)`
: タグ *tag* と共に宣言されたレコードの項目番号 *field* が比較対象外である場合、 `true` を返す。

結果型 : **bool**

`AskDontCare(int tag, int field)`

: タグ *tag* とともに宣言されたレコードの項目番号 *field* が比較対象外であるならば、 `true` を返す。

結果型 : **bool**

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

: 項目番号 `field` をタグ番号 *tag* をもつレコードに対する比較対象外として、印をつける。

結果型 : **void**

`SetDontCare(int tag, int field)`

項目番号 `field` をタグ番号 *tag* をもつレコードに対する比較対象外として、印をつける。

結果型：**void**

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

タグ番号 *tag* の象徴タグ (レコードの文字列タグ) を *symtag* に設定する。これは *VDMRecInfoMap* において、*tag* を *symtab* に関連付ける。レコード値を出力するために *ascii* メソッドを用いるとき、もし特定タグ番号に対して象徴タグが定義されているのならば、象徴タグ文字列がタグ番号 *tag* の代わりに印刷されることになる。

結果型：**void**

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

タグ番号 *tag* の象徴タグ (レコードの文字列タグ) を *symtag* に設定する。これは *VDMRecInfoMap* において、*tag* を *symtab* に関連付ける。レコード値を出力するために *ascii* メソッドを用いるとき、もし特定タグ番号に対して象徴タグが定義されているのならば、象徴タグ文字列がタグ番号 *tag* の代わりに印刷されることになる。

結果型：**void**

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

この関数で、ある関数へのポインタをタグ番号 *tag* に関連付けることが可能である。関数 *f* は、タグ *tag* をもつレコード上の *ascii* メソッドを呼び出すときに用いられる。

結果型：**void**

`SetPrintFunction(int tag, vdm_pp_function_ptr f)`

この関数で、関数 *f* へのポインタをタグ番号 *tag* へ関連付けることが可能である。関数 *f* はタグ *tag* をもつレコード上の *ascii* メソッドを呼び出すときに用いられる。

結果型：**void**

`GetSize(int tag)`

タグ番号 *tag* と共に宣言されたレコードのサイズを返す。

結果型：**int**

`GetSymTag(int tag, wstring & s)`

象徴文字列タグを抽出し、レコード *tag* のワイド文字列 *s* に代入する。

結果型：**bool**

`size()`

写像 *VDMRecInfoMap* のサイズを返す。

結果型：**int**

`dump(ostream & o)`

VDMRecInfoMap 内の情報を出力ワイドストリーム *o* に印刷する。

結果型：**void**

5.13 Tuple

`Tuple` は以下の要素関数をサポートする:

`Tuple Tp`

`Tp` を `Tuple` 型の値として宣言する。項目数は 0 とする。

結果型：**void**

`Tuple Tp(i)`

`Tp` を `i` 項目もった `Tuple` 型の値として宣言する。

結果型：**void**

`Tuple Tp(Tp1)`
: `Tp` を、`Tp1` と等しい `Tuple` 型の値として宣言する。

結果型：**void**

`Tp.SetField(i,A)`
: `i` 番目の項目を修正して `A` とする。`i` が `Tp` に対して定義された項目数以内でなければ、エラー 第 7 章参照 が起こされる。関数は `Tp` への参照を返す。

結果型：**Tuple&**

`Tp.GetField(i)`
: `Tp` の `i` 番目の項目の内容 `G` を返す。`i` が `Tp` に対して定義された項目数以内でなければ、エラー 第 7 章参照 が起こされる。

結果型：**Generic&**

`Tp.Length()`
: `Tp` に対して宣言された項目数を返す。

結果型：**int**

例題

```
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;
```

このプログラムの実行結果は次の通り：

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

5.14 ObjectRef

ObjectRef は、VDM^{++} [3]、*VDM++ Language Manual*[1] においてオブジェクト参照型を実装するために用いられる。このクラスは、 *VDM++ to C++ Code Generator* [2] によって生成されたクラスのインスタンスの参照を含めるためにのみ用いられるべきである。

ObjectRef は、 C++クラスのインスタンスへの参照 (ポインタ) と、参照ポインタにより指定されたインスタンスの型を識別する型変数、を含む。

VDM C++ ライブラリの実装は参照カウンタ上に基づくため、クラス `ObjectRef` の全てのオブジェクトから参照されない場合は、クラスのインスタンスへのポインタは削除される。

`ObjectRef` は以下の要素関数をサポートする:

`ObjectRef Ob (p)`
: `Ob` を `ObjectRef` 型の値として宣言する。参照ポインタは `p` に設定される。参照ポインタは C++ `new` 演算子と共に生成されていなければならない。これは組込ガベージ収集処理を正しく働かせるために必要である。もしポインタが指定されていないならば、`NULL` ポインタが既定のパラメータとして用いられる。

  結果型 : **void**

`ObjectRef Ob(Ob1)`
: `Ob` を、 `ObjectRef` 型の値で `Ob1` と等しいものとして宣言する。

  結果型 : **void**

`Ob.MyObjectId()`
: `Ob` の型 `i` を返す。型は `CGBase.h` で列挙型として定義された整数である。

  結果型 : **int**

`Ob.GetRef()`
: `Ob` の vdmBase ポインタ `p` を返す。

  結果型 : **vdmBase***

`Ob.SameBaseClass(Ob1)`

`Ob` と `Ob1` が同じ基本クラスをもつならば `True` を返す、つまりもし `Ob` と `Ob1` が、同じルートのスーパークラスからの派生が可能であるクラスのインスタンスであるならば `True` を返し、それ以外は `False` を返す。

結果型：**Bool**

`Ob.IsOfClass(i)`

`Ob` が型 `i` のクラスの、あるいは `i` の任意のサブクラスのオブジェクトを参照するならば `True` を返し、そうでない場合は `False` を返す。`i` は `MyObjectId` で返された型である。

結果型：**Bool**

`Ob.IsOfBaseClass(i)`

型 `i` のクラスが、`Ob` により参照されるオブジェクトの継承連鎖で、ルートのスーパークラスであるならば `True` を返し、他の場合は `False` を返す。

結果型：**Bool**

`Ob.ascii()`

`ObjectRef` に対する `ascii()` メソッドの結果は、型項目と参照ポインタ (hex 形式) を返す。

5.14.1 CGBase

ObjectRef はコード生成されたクラス CGBase に関連している。このクラスは、派生したものでない VDM++ スーパークラスに対するスーパークラスとなる。加えて、このクラスはオブジェクト参照からコード生成されたクラスへのポインタへのキャスト関数や、コード生成された 各VDM++ クラスの一意的なタグ付けに用いられる列挙型も定義されている。これはヘッダファイル `CGBase.h` で定義され、 `CGBase.cc` で実装されている。

`ObjGet_`***class-name***`(Ob)`

コード生成されたクラス *class-name* に対するポインタを返す。`Ob` が *class-name* を参照していない場合にはゼロを返す。

結果型：***class-name ****

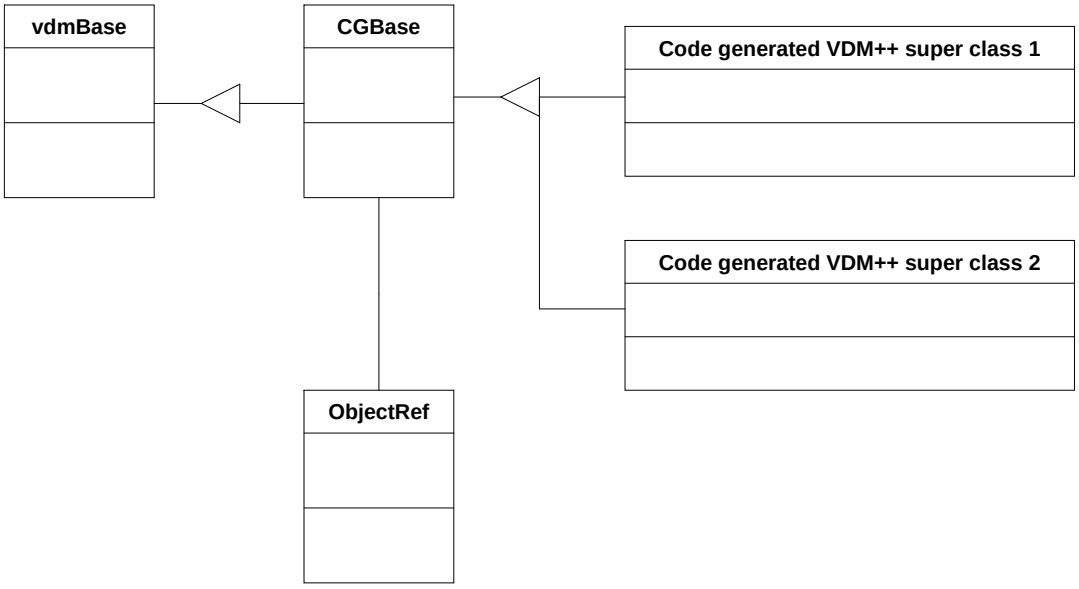

図 3: C++ クラス間の関係

例題

以下の VDM^{++} クラスを考えよう:

```
class A

operations
  public Test: () ==> nat
  Test()  ==
    let a = 10 + 10 in
      return a

end A
```

以下のヘッダーファイル、 A.h、は *VDM++ to C++ Code Generator* によって生成されている:

```
#ifndef _A_h
```

```
#define _A_h

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

class type_ref_A : public virtual ObjectRef {
public:
  type_ref_A () : ObjectRef() {}
  type_ref_A (const Generic &c) : ObjectRef(c) {}
  type_ref_A (vdmBase * p) : ObjectRef(p) {}

  const wchar_t * GetTypeName () const {
  return L"type_ref_A";
  }
};

class vdm_A : public virtual CGBase {
friend class init_A ;
public:

vdm_A * Get_vdm_A () {
  return this;
}

ObjectRef Self () {
  return ObjectRef(Get_vdm_A());
}

int vdm_GetId () {
  return VDM_A;
};
```

```
vdm_A ();
virtual ~vdm_A ()
public:
  virtual Int vdm_Test ();
};

#endif
```

これで以下の方法により、 A 型のオブジェクト参照を実装するために ObjectRef を用いることができる:

```
#include <iostream>
#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;

  return 0;
```

}

このプログラムの実行結果は次の通り：

```
0 is the value of VDM_A
[ 10,@(0, 0x808dde4),@(0, 0x808debc) ] 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 は以下の要素関数をサポートする:

Generic G
: Generic 型の値として G を宣言する。値は GenericVal のインスタンスである。

  結果型：**void**

Generic G(A)
: G を、 A の値と等しい Generic 型の値として宣言する。

  結果型：**void**

6 集合型、列型、写像型

集合、列、写像の型に対しては、相当する C++ テンプレートがある。これらテンプレートを用いることで、さらに適切な型情報の型宣言を行うことが可能となる。これらの型を用いると、集合だけでなく集合が含めることが可能な値型も宣言することができる。

6.1 集合型

SET テンプレートは Set クラスから派生する。

SET テンプレートはコンストラクタ関数をサポートする:

SET<A> St
: St を A 型の Set として宣言する。 St の値は空集合に初期化される。

結果型: **void**

SET<A> St(St1)
: St を A 型の Set として宣言する。 St の値は St1 の値に初期化される。

結果型: **void**

加えて Set クラスに作用する同等の関数や操作も SET テンプレートに対して宣言される。

6.2 列型

SEQ テンプレートは Sequence クラスから派生する。

SEQ テンプレートはコンストラクタ関数をサポートする:

SEQ<A> Sq
: Sq を A 型の Sequence として宣言する。 Sq の値は空列に初期化される。

結果型: **void**

SEQ<A> Sq(Sq1)
: Sq を A 型の Sequence として宣言する。 Sq の値は Sq1 の値に初期化される。

結果型: **void**

加えて、 Sequence クラスに作用する同等の関数や操作も SEQ テンプレートに対して宣言される。

6.3 写像型

MAP テンプレートは Map クラスから派生する。

MAP テンプレートはコンストラクタ関数をサポートする:

MAP<A,B> M
: M を A 型から B への Map として宣言する。 M の値は空写像に初期化される。

結果型: **void**

MAP<A, B> M(M1)
: M を A 型から B への Map として宣言する。M の値は M1 の値に初期化される。

結果型: **void**

加えて、 Map クラスに作用する同等の関数や操作も MAP テンプレートのために宣言されている。

7 エラーメッセージ

エラーがライブラリ関数によって検出された場合、m4err ストリームに、エラーを検出したライブラリ関数を記述する文字列と共にエラー番号が 書かれる。その後プログラムを終了するために 'exit' 関数が呼び出される。

エラーは 2 つのカテゴリに分けられてきた。ユーザーエラー (U) はライブラリの通常使用の下でおきる可能性がある。例としては、空集合から要素を抽出しようとした場合である。

内部エラー (I) はさらに重大なエラーである。これらのエラーは、ライブラリの通常使用の下では現れるはずのないものである。

No.	記号名	説明	エラー型
1	ML_CONFLICTING_RNGVAL	異なる値域値が既に存在する写像へのキーの挿入	U
2	ML_NOT_IN_DOM	写像に対し定義域中にないキーの適用	U
3	ML_CAST_ERROR	汎用型の誤った型へのキャスト	U
4	ML_INDEX_OUT_OF_RANGE	列、組、レコードの関数における範囲外のインデックス	U
5	ML_OP_ON_EMPTY_SEQ	空列に対する違法な関数呼び出し	U
6	ML_OP_ON_EMPTY_SET	空集合に対する違法な関数呼び出し	U
7	ML_NOT_IN_SET	集合に存在しない要素を取り除こうとした	U
8	ML_ASSIGN_ERROR	異なる型の 2 変数間で代入をしようとした	U
9	ML_TRAVERSE_CONFLICT	要素関数 Next を評価しているときに検出されたエラー	I
10	ML_HD_ON_EMPTY_SEQUENCE	空列で hd を取り出そうとした	U
11	ML_TL_ON_EMPTY_SEQUENCE	空列で tl を取り出そうとした	U
12	ML_RANGE_ERROR	組やレコードに対する範囲外のインデックス	U
13	ML_ZERO_REFCOUNT	ゼロ参照カウンタの検出	I
14	ML_NULL_REF	ゼロポインタ参照	I
15	ML_DIV_BY_ZERO	ゼロによる除算	U

参考文献

[1] CSK. *The VDM++ Language*. CSK.

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

[3] 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 ファイル

VDM C++ ライブラリへのインターフェイスは以下のファイルから構成される:

metaiv.h は、第 5 章で述べたすべての型の特有関数のプロトタイプを含むヘッダーファイルである。

vdm.lib (Windows) は、本書で述べた全関数の実装を含むアーカイブライブラリである。

libvdm.a (Mac、Linux) は、本書で述べた全関数の実装を含むアーカイブライブラリである。