

# VDMTools

---

C++コード生成マニュアル  
(VDM++)  
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](mailto:inq@fmvdm.org)

Mail

*C++コード生成マニュアル (VDM++) 1.0*

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

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

## 目 次

1	導入	1
2	コードジェネレータの起動	2
2.1	コード生成のための要件	2
2.2	グラフィカルインターフェイスの利用	2
2.3	コマンドラインインターフェイスの使用	6
2.4	C++ 生成ファイル	6
3	生成コードとのインターフェイス接続	7
3.1	VDM++ 型のコード生成 - 基本	8
3.2	ユーザーにより実装されたファイル	12
3.2.1	レコードタグに用いるオフセットの定義	13
3.2.2	陰関数 / 陰操作および宣言文の実装	14
3.2.3	メインプログラムの実装	15
3.2.4	生成された C++ コードの部分的置き換え	18
3.3	C++コードのコンパイル、リンク、実行	19
4	サポートされていない構成要素	21
5	VDM 仕様のコード生成 - 詳細	23
5.1	生成コードクラス	23
5.1.1	VDM C++ ライブラリにおけるオブジェクト参照	23
5.1.2	クラスの生成コードの継承構造	25
5.1.3	生成されたクラスの構造	29
5.2	型のコード生成	30
5.2.1	動機付け	30
5.2.2	VDM++ 型から C++へのマップ	33
5.2.3	VDM++ 型名称のコード生成	38
5.2.4	不変条件	41
5.3	関数定義と操作定義のコード生成	41
5.4	インスタンス変数のコード生成	45
5.5	値定義のコード生成	47
5.6	式と文のコード生成	48
5.7	名称仕様	48
5.8	標準ライブラリ	49



<b>A</b>	<b>libCG.a ライブラリ</b>	<b>50</b>
A.1	cg.h . . . . .	50
A.2	cg_aux.h . . . . .	52
<b>B</b>	<b>C++ 手書きファイル</b>	<b>53</b>
B.1	DoSort_userdef.h . . . . .	53
B.2	ExplSort_userdef.h . . . . .	53
B.3	ImplSort_userdef.h . . . . .	53
B.4	MergeSort_userdef.h . . . . .	53
B.5	SortMachine_userdef.h . . . . .	54
B.6	Sorter_userdef.h . . . . .	54
B.7	ImplSort_userimpl.cc . . . . .	54
B.8	sort_pp.cc . . . . .	55
<b>C</b>	<b>Make ファイル</b>	<b>58</b>
C.1	Unix プラットフォームのための Make ファイル . . . . .	58
C.2	Windows プラットフォームのための Make ファイル . . . . .	63

# 1 導入

VDM++ to C++ コードジェネレータは、VDM++ 仕様から C++ コード自動生成を行うための支援をする。これにより、コードジェネレータは VDM++ 仕様に基づいたアプリケーションの実装を、速い方法で提供する。

コードジェネレータは VDM++ Toolbox に対しアドオン形式をとる。このインストールについては、文書 [2] に記載されている。本書は *VDMTools* ユーザマニュアル ( *VDM++* ) [5] の拡張であり、VDM++ to C++ コードジェネレータへの導入を行う。

コードジェネレータは、VDM++ 構成要素全体のおよそ 95% をサポートする。捕捉として、生成コードの一部を手書きコードに置き換えることがユーザーに許されている。

本書は以下のように構成されている:

第 2 章で、VDM++ 仕様が正しい C++ コードを生成するために満たされなければならない要件を並べる。さらにこの章で、VDM++ Toolbox から C++ ファイルを起動する方法を述べる。最後に、コード生成された C++ ファイルを記載する。

第 3 章は、インターフェイスを記述しながら C++ 生成コードへの導入を行い、そしてそれをどのように手書きコードと結び付けるかを説明する。さらに、C++ コードのコンパイル、リンク、実行の方法まで説明する。

第 4 章では、コードジェネレータでサポートされない VDM++ 構成要素をまとめて示す。

第 5 章は、C++ 生成コード構造の詳細を記載する。加えて VDM++ と C++ のデータ型間の関連を説明し、VDM++ to C++ コードジェネレータの開発中に用いる名称仕様を含めて、設計上決められている事柄をいくつか述べる。ここは、コードジェネレータを職業的に使用する前には集中的に学習すべき章である。

## 2 コードジェネレータの起動

コードジェネレータの使用を始めるため、1つ以上のファイルにわたったVDM++仕様を書いておくべきであろう。Toolboxの配布中に、様々なソートアルゴリズムの仕様が含まれている。この仕様は、以下においてコードジェネレータの使用を記述するために用いられることとなる。これは[4]で述べられている。この手順1つ1つを、自身のコンピュータ上で確認していくことが推奨される。そのため、ディレクトリ `vdmhome/examples/sort` をコピーしそこへ `cd` にて移動すること。

C++コードを生成する前に、VDM++仕様が必要な要件を満たしているかどうか確認する必要がある。この要件は第2.1章に記述されている。第2.2章と2.3章では、VDM++ Toolboxを用いてグラフィカルインターフェイスやコマンドラインからC++コードを生成する方法を説明する。第2.4章では、コード生成されたC++ファイルを記載する。

### 2.1 コード生成のための要件

コードジェネレータが正しいコード生成を行うために、VDM++仕様の全ファイルは構文チェックされることが要件となる。つまり1クラスのコード生成であっても、それを可能とするためには仕様中全ファイルのチェックが必要となる。

さらにコードジェネレータは、適正な型のクラスに対してしかコード生成を行うことができない。<sup>1</sup>あるクラスに対して事前に型チェックを行わずコード生成しようとした場合は、Toolboxが自動的に型チェックを行う。


### 2.2 グラフィカルインターフェイスの利用

ここではVDM++ Toolboxのグラフィカルユーザーインターフェイスから、どのようにソート例題をコード生成するかを述べていく。

---

<sup>1</sup> [3]で説明されるが、2つのよく形式化されたクラスが存在する。ここではできるだけ十分によく形式化された型の適正さを意味する。

VDM++ Toolbox は `vdmgde` コマンドで始める。ソートの例題に相当するコードを生成するには、`/vdmhome/examples/sort` ディレクトリに置かれたすべての `*.rtf` ファイルを含めて新しいプロジェクトを生成することである。プロジェクトの構成法の記載としては、[5] を参照のこと。

最初はファイルに対して構文チェックと型チェックを行う必要がある：手作業で行わない場合には、コードジェネレータが起動されるとき Toolbox が自動的にこれを行うはずである。ソート例題の仕様では、エラーなしで両チェックを通過する。その後すべてのクラスを選択して、 (Generate C++) ボタンを押すことで、コードジェネレータが起動可能となる。1つ以上のファイルあるいはクラスを選択することができ、この場合にはこれらが C++ コードに翻訳される。この手順の結果が 図 1 に示されている。

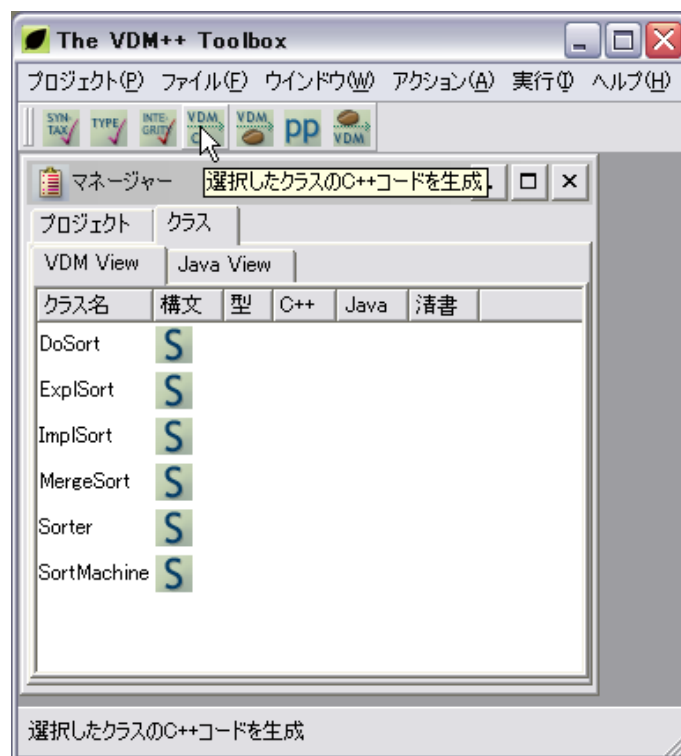


図 1: ソート例題のコード生成

仕様中の各 クラス に対してコードが生成されると、図 2 に見られるように Toolbox は大きな *C* を書き入れてこれを知らせる。プロジェクトファイルのおかれたディレクトリ中で、たくさんの C++ ファイルが生成される。プロジェクトファイルの存在しない場合には、これらのファイルは VDM++ Toolbox が始め

られたディレクトリに書かれることになる。

ソート例題に対してコード生成を行う時、コードジェネレータによって警告が1つ生成されるため、図 3 のような *Error* ウィンドウが出る。

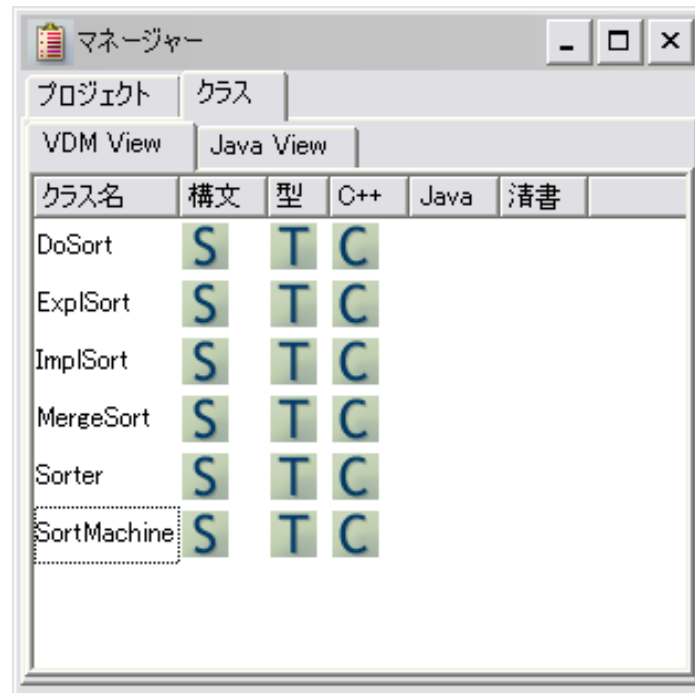


図 2: ソート例題のコード生成

この警告は、コードジェネレータでは連結パターンがサポートされないことを示している。コードジェネレータはこの構成要素に対して、実行可能な C++ コード生成を行うことができないということを意味する。生成されるコードはコンパイル可能なものであるが、サポートされない構成要素が含まれる枝葉部分の実行は実行時エラーを引き起こす。第 4 章で、サポートされない構成要素の詳細リストを提示する。

コードジェネレータのユーザーは、実行時エラーの位置情報を含めたコード生成を選択できる。出力位置情報 オプションが選択されていると、実行時エラーの原因である VDM++ 仕様における位置 (ファイル名および行番号と列番号) を実行時エラーメッセージで伝えてくれる。図 4 で示すように、オプションメニューでこの機能は設定できる。もう1つ利用可能なオプションとして、事前事後条件チェックオプションがある。これは、操作や関数の事前条件や関数の事後条件をチェックするコードを生成する。これも図 4 で示している。





図 3: コードジェネレータで生成された警告



図 4: C++ コードジェネレータのオプション

ソート例題に対しては MergeSort 関数の実行が、ここで述べた実行時エラーを引起す。記載したオプション設定を行わなかった場合、相当する C++ コードの実行で次のエラーメッセージが出る:

```
The construct is not supported: Sequence concatenation pattern
```

一方、オプションを設定した場合は次のようなエラーメッセージとなる:

```
Last recorded position:
```

```
In: MergeSort. At line: 31 column: 18
```

```
The construct is not supported: Sequence concatenation pattern
```

## 2.3 コマンドラインインターフェイスの使用

VDM++ Toolbox をコマンドラインから実行する場合にも、もちろんコードジェネレータ を起動できる。これは以下に簡単に述べられている。

VDM++ Toolbox は、コマンドラインから `vppde` コマンドで始める。コードを生成するために `-c` オプションを用いる:

```
vppde -c [-r] [-P] specfile, ...
```

ソート例題のコード生成を行うために、次のコマンドを `vpphome/examples/sort` ディレクトリで実行する:

```
vppde -c *.vpp
```

仕様は最初に構文解析がなされる。構文エラーが検出されない場合、できる限り適正となるように型チェックがなされる。型エラーが検出されなかった場合には、仕様は変換され最後にたくさんの C++ ファイルになる。グラフィカルインターフェイスに相当するものとしては、実行時の位置情報をもつコード生成のために出力位置情報 オプション (`-r`) を設定することができ、さらに事前事後条件の実行時チェックを行うための 事前事後条件チェック オプション (`-P`) が設定できる。

## 2.4 C++ 生成ファイル

さらに1つ手順を進めて、コードジェネレータにより生成されたファイルを見てみよう。VDM++ クラスの各々に対して4つのファイルが生成される:

- `<ClassName>.h`
- `<ClassName>.cc`
- `<ClassName>_anonym.h`
- `<ClassName>_anonym.cc`

<ClassName>.h ファイルは、VDM++ クラスに相当する C++ クラスの定義を含む。さらに、VDM++ クラスで定義された合成型に相当するクラス定義を含む。

<ClassName>.cc ファイルは、VDM++ クラスで定義された関数と操作の実装を含む。さらに、レコード型に対して生成される全クラスの要素関数の実装がここにある。

<ClassName>\_anonym.h ファイルと <ClassName>\_anonym.cc ファイルは、全匿名型の宣言と実装を行うためにある。匿名型とは、VDM++ 仕様において名前が与えられていないものである。

これらのファイルとは別に、さらに2つのファイルが生成される：

- CGBase.h
- CGBase.cc

これらのファイルはオブジェクト参照型の一部の実装を含む。これは第 5.1 章で述べる。

各 VDM++ クラスに相当するコードは、ヘッダーファイルと実装ファイルに分けられる。両ファイルには クラス名称が与えられる。ヘッダーファイルの拡張子は '.h' となる一方、実装ファイルの拡張子は、Unix プラットフォーム上では '.cc'、Window プラットフォーム上では '.cpp' となる。実装ファイルの拡張子は環境変数の設定により VDMCGEXT<sup>2</sup>、カスタマイズすることができる。

### 3 生成コードとのインターフェイス接続

ここまでで、1つの VDM++ 仕様からたくさんの C++ ファイルが生成されるところまで到達した。アプリケーションのコンパイル・リンク・実行を行うためには、これらの C++ ファイルに対してインターフェイスを書くことになる。

生成コードに対しインターフェイスの記述を行うためには、生成コードについてのいくつか基本的な知識が必要となる。これはまず最初に、VDM++ 構成要素、

---

<sup>2</sup>Windows 2000/XP/Vista 上ではレジストリに設定できる

特に VDM++ 型に対してコード生成を行う時に、用いる方策を含める。以下でこのことについて簡単な導入を行う。さらなる情報は第 5 章に記載する。

### 3.1 VDM++ 型のコード生成 - 基本

この章では、VDM++ の型をコード生成する方法について簡単な導入を行う。

まずは C++ 生成コードの例題を提示することから始める。次は関数 `IsOrdered` のシグニチャである

```
IsOrdered: seq of int -> bool
```

`ExplSort` クラスで定義されていて、以下のようにコード生成される:

```
class type_iL : public SEQ<Int>{
...
};

Bool vdm_ExplSort::vdm_IsOrdered(const type_iL &vdm_l) {
...
};
```

このコードを理解するため、VDM 型に対してコード生成を行うための方策を知ると共に、同様に用いられる名称仕様についても、いくつか知識が必要となる。

コードジェネレータのデータ型の扱いは、VDM C++ ライブラリに基づく。このライブラリ (`libvdm.a`) の最新版を [\[1\]](#) に記載する。

- 基本データ型

基本データ型は、相当する VDM C++ ライブラリクラスである、`Bool`、`Int`、`Real`、`Char`、`Token` にマップされる。

- 引用型

引用型は、相当する VDM C++ ライブラリクラス `Quote` にマップされる。

- 集合、列、写像型

合成型 `set`、`sequence`、`map`、を取り扱うために、テンプレートが導入されている。これらテンプレートは VDM C++ ライブラリにおいても定義されている。例として、VDM 型である `seq of int` がどのようにコード生成されるかを見よう:

```
class type_iL : public SEQ<Int>{  
    ...  
};
```

VDM `seq` 型は、テンプレートである `SEQ` クラスから継承されるクラスにマップされる。`seq of int` の場合、テンプレートクラスの引数は `Int` であり、これは基本的な VDM 型 `int` を表す C++ クラスである。新しいクラスの名称は以下のように作られている:

type: 匿名型を示す  
i: 整数を示す  
L: 列を示す

- 合成型/レコード型

各合成型はマップされて、VDM C++ ライブラリ `Record` クラスのサブクラスにマップされる。たとえば次の クラス `M` で定義された次の合成型は

```
A:: r : real  
    i : int
```

以下のようにコード生成される:

```
class TYPE_M_A : public Record{  
    ...  
};
```

- 組型

組の取り扱いは、合成型に対するものと大変よく似ている。各組型は、VDM C++ ライブラリ `Tuple` クラスのサブクラスにマップされる。たとえば、次の組は:

```
int * real
```

以下のようにコード生成される:

```
class type_ir2P : public Tuple{  
    ...  
};
```

新しいクラス名称は次の方法で作成される:

type: 匿名型を示す  
i: 整数を示す  
r: 実数を示す  
2P: 次数 2 の組を示す

- ユニオン型

ユニオン型は、VDM C++ ライブラリ Generic クラスにマップされる。

- オプション型

オプション型はマップされ VDM C++ ライブラリの Generic クラスとなる。

- オブジェクト参照型

各 VDM++ クラスに対して相当する C++ クラスが生成される。VDM++ クラス *SortMachine* に対しては、相当する C++ クラス *vdm\_SortMachine* が生成される。

クラスインスタンスのオブジェクト参照は、クラス `type_ref_<ClassName>` にマップされる。

オブジェクト参照型のことは第 5.1 章で詳細に述べる。

合成型の例において、既に型名称の生成方法について構想が与えられている。

仕様において名称が与えられていない型 (匿名型) には、小文字の `type` が前につけられる。しかし型名称には大文字で `TYPE` が前に付けられる。既にレコード例題で、`TYPE_M_A` と名づけられた型名称の例を見てきた。他の VDM++ 型ももちろん名付けることができ、名称スキームはレコード型に対して用いたのと同じである。

生成される型名称には、TYPE が前に付けられる。その後 クラス名が続き、ここで型が定義され、最後に選択された VDM 名称が連結される。

以下の VDM++ 仕様と定義型の生成では、用いられた名称仕様を見よう:

```
class M
types
  A = int;
  B = int * real;
  C = seq of int;
end M
```

上記 3 つの定義型には、名称 TYPE\_M\_A、TYPE\_M\_B、TYPE\_M\_C が与えられる。これら型名称のスコープは クラス M に限定される。したがってこれらの名称定義は、ファイル *M.h* に置かれる。

```
#define TYPE_M_A Int
#define TYPE_M_B type_ir2P
#define TYPE_M_C type_iL
```

しかし、仕様はまた 2 つの匿名型 `int * real` と `seq of int` も含める。これらの型は潜在的にどのような クラスでも用いることができ、したがって相当する C++ 型の名称と定義はグローバルに<sup>3</sup>宣言および定義されるべきである。これは *anonym* ファイル内でなされている: `<ClassName>_anonym.{h,cc}`

型がコード生成される方法について与えられる情報に加え、どのように関数や操作の名称が生成されるかについて触れておくべきだ: VDM 仕様内の M クラスにおける関数あるいは操作の名称 *f* に対し、次の名称が与えられる: `vdm_M::vdm_f`.

ここで VDM 仕様に対してコード生成を行う際の コードジェネレータ の全体的方策を把握しておくべきであろう。さらに詳細な情報は第 5 章で述べるので、コードジェネレータ を専門的に用いる場合は丁寧に学習する必要がある。

---

<sup>3</sup>構造的に等しい型に対して単一の名称を定義するという方策である。この方策は C++ は型名称等価に基づくという現実に対処するために用いられ、一方で VDM++ は構造等価に基づいている。

### 3.2 ユーザーにより実装されたファイル

コードジェネレータ およびコード生成されたファイルについて、基本的な情報はここまでですでに述べた。この章では、生成コードとインターフェイスをとるために行わなければならない作業を述べる。

まず、VDM++ 仕様に対する C++コードを実行する場合に含まれるすべての C++ ファイルについて、概観を示すことから始めよう。これらのファイルは、コード生成される C++ ファイルと手作業でコーディングされる C++ ファイルに分けることができる。図 5 では、コード生成ファイルを左に、手書きファイルを右に示している。さらにそれらに含まれるファイルを、VDM++ クラス単位の C++ ファイルと VDM++ 仕様単位の C++ ファイルに分けることができる。

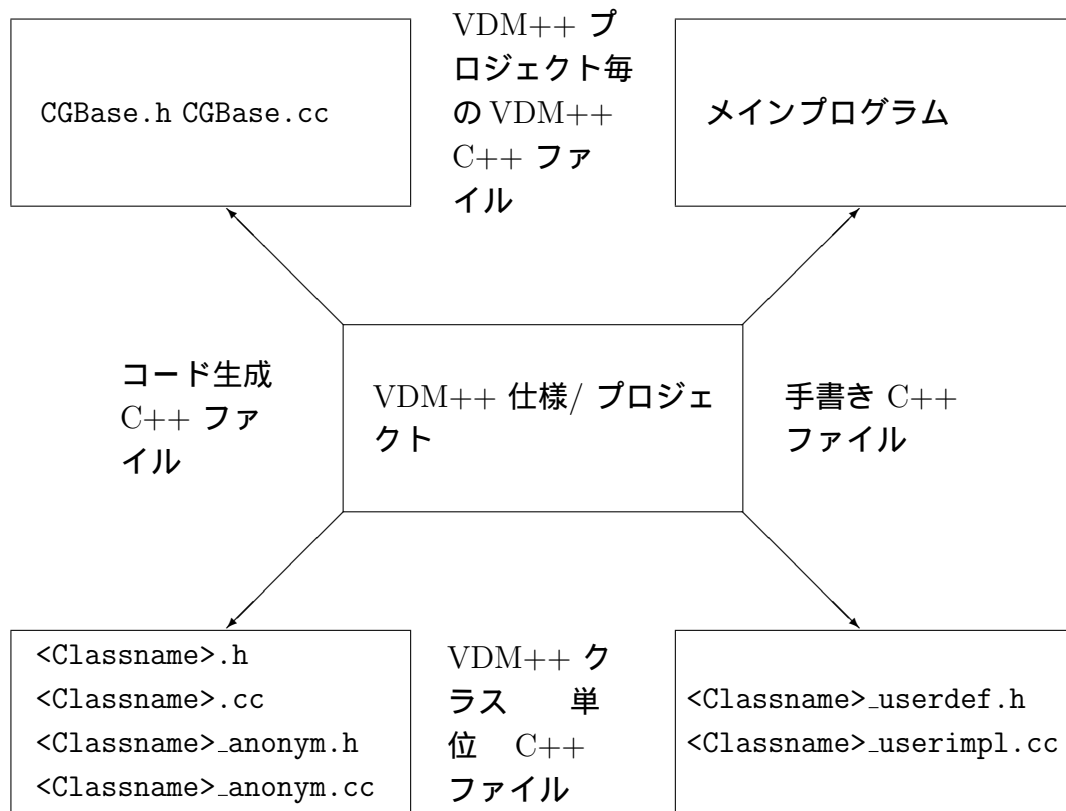


図 5: VDM++ プロジェクト単位の C++ファイル

第 2.4 章ですでに、コード生成された C++ ファイルについて述べた。ここでは手書きコードファイルについて述べよう。



生成コードとインターフェイスをとるために、以下の作業の完了が必要である:

1. 各 クラスに対して、レコードタグに用いるオフセットを定義する。
2. VDM++ 仕様に含まれている陰関数 / 陰操作および宣言文を実装する。
3. メインプログラムを書く。
4. 選択的に、生成された C++ コードを部分的に手書きコードと置き換える。
5. アプリケーションのコンパイル、リンク、実行を行う。

以下ではソート例題について、これらの作業を順に 1 つ 1 つ説明していく。

### 3.2.1 レコードタグに用いるオフセットの定義

VDM++ 仕様において合成型 (レコード型) は、文字列 (タグ) とレコード型の各項目に対する選択項目の並びで構成される:

```
RecTag ::  
  fieldsel1 : nat  
  fieldsel2 : bool
```

VDM C++ ライブラリでは、レコードタグ文字列 “*RecTag*” が単一整数を用いてモデル化されている。しかしこの方策をとるためには、1 つの仕様に含まれる全レコード型が各々単一のタグ番号を有していることが重要である。コードジェネレータ は各クラス に対し、オフセットを基として連続的にレコードタグ番号を振り当てる。オフセットはユーザーによって定義されるべきものであり、各 クラス に対し定義されるオフセットがタグの唯一性を保証するものとなるかどうかは、ユーザーの責任である。

オフセット定義は、<ClassName>\_userdef.h という名のファイル中に書き込まれるべきである。オフセットは定義命令で定義し、タグは TAG\_<ClassName>. という名称とするべきである。

ソート例題においてユーザーは、各クラスに 1 つで 6 つのファイルを実装する必要がある。MergeSort\_userdef.h ファイルには、たとえば次のような定義が含まれるだろう。

```
#define TAG_MergeSort 100
```

### 3.2.2 陰関数 / 陰操作および宣言文の実装

陰関数を含むすべての クラス に対して、その関数定義を含んだ `<ClassName>_userimpl.cc` というファイルが書かれる必要がある。

ソート例題は `ImplSort` クラス内に、`ImplSorter` という陰関数を 1 つ含む。生成された C++ コードのインターフェイスのために、この関数は `ImplSort_userimpl.cc` ファイル中に実装されていなければならない。この関数は `ImplSort.h` ファイル中に定義されているもので、`vdm_ImplSort` クラス中に見つかるその要素関数宣言と一致するように書かれていなければならない:

```
virtual type_iL vdm_ImplSorter(const type_iL &);
```

クラス `ImplSort` 中の関数 `ImplSorter` は `vdm_ImplSort::vdm_ImplSorter` という名称となり、以下のように宣言される:

```
type_iL vdm_ImplSort::vdm_ImplSorter(const type_iL& l) {  
    ...  
}
```

この関数の実装の一例を 付録 B.7 に示してある。

このようにユーザーは、操作に対して C++ 関数定義を書き、それを含む クラスを `<ClassName>_userimpl.cc` ファイルに加えなければならない。

コードジェネレータは、出会った宣言文の各々に対しインクルード命令を作成する。このようにユーザーは、各仕様文に対して相当する名称のファイル: `vdm_<ClassName>_<OperationName>-<No>.cc`, を実装する必要があるが、ここにおける `<OperationName>` は宣言文が現れる操作の名称で、`<No>` は操作仕様にある仕様文に連続番号をふったものとなる。

### 3.2.3 メインプログラムの実装

ここまで、メインプログラムを除くコードのコンパイル、リンク、実行に必要なファイルの実装を行ってきた。

ここで、ソート例題に対するメインプログラムを書いてみよう。

まず最初に、VDM++のメインプログラムの仕様を定めることから始めることとする。

```
01  Main: () ==> ()
02  Main () ==
03      let arr1 = [3,5,2,23,1,42,98,31],
04          arr2 = [3,1,2] in
05
06      ( dcl smach : SortMachine := new SortMachine(),
07          res : seq of int = [] ;
08          def dos : Sorter := new DoSort() in
09              res = smach.SetAndSort(dos,arr1);
10          def expls : Sorter := new ExplSort() in
11              res = smach.SetAndSort(expls,arr2);
12          def imps : Sorter := new ImplSort() in
13              ( res = smach.SetAndSort(imps,arr2)
14                  imps.Post_ImplSorter(arr2,res)
15              )
16          def mergs : Sorter := new MergeSort() in
17              smach.SetSort(mergs);
18          res = smach.GoSorting(arr2);
19      )
```

そして上記仕様の、VDM++ メソッドと同等の機能を有するC++のメインプログラムを実装していく。メインプログラムは `sort_pp.cc` に実装されていて、全プログラムを 付録 B.8 で見ることができる。

メインプログラムを含めたC++ファイルは、すべての必要なヘッダーファイルをすべてインクルードすることから始めるべきである。これらは、各 VDM++

クラスごとのヘッダー、VDM C++ ライブラリ のヘッダー、metaiv.h、そして (出力を生成するための) 標準ライブラリクラス <fstream> を含める。

ここでは、上記に列挙した VDM 仕様を順次 C++へ翻訳していこう。行 03 と 04 で 2 つの整数リストが明記される。C++に翻訳されると次のコードとなる:

```
type_iL arr1, arr2;
arr1.ImpAppend ((Int)3);
arr1.ImpAppend ((Int)5);
...
arr2.ImpAppend ((Int)3).ImpAppend ((Int)1).ImpAppend ((Int)2);
```

行 06 は、SortMachine クラスのインスタンスに対するオブジェクト参照 smach を宣言している。

次の行はこれを C++で実装する:

```
type_ref_SortMachine smach (ObjectRef (new vdm_SortMachine ()));
```

行 07 は seq of int 型の変数 res を宣言していて、これは後にソートされた整数列を含めるために用いられる。これに対する C++ コードは次の通り:

```
type_iL res;
```

ここでどのようにして仕様のソートメソッドを呼び出すかを見よう。

VDM++ 仕様と生成された C++コードから分かるように、SortMachine クラスはインスタンス変数として Sorter 抽象型クラスに対するオブジェクト参照をもつ。Sorter のサブクラスは異なるソートアルゴリズムを実装している。SortMachine クラスの SetAndSort メソッドは 2 つのパラメータをもつ: Sorter のサブクラスのインスタンスと整数の列である。メソッドは、前述のインスタンス変数が特定のサブクラスしたがって特定のソートアルゴリズムを参照するように設定し、その後このクラスの Sort メソッドを整数列をパラメータとして呼び出す。結果はソート済みの整数列となるはずである。

行 08 は、DoSort クラスのインスタンスに対するオブジェクト参照 dos を宣言し、SortMachine クラスの SetAndSort メソッドを宣言されたオブジェクト参照 dos と整数列 arr1 を引数として呼び出す。結果は res で与えられる。

すべてのコード生成された VDM++ 型は `ascii` メソッドをもち、それぞれの VDM 値の ASCII 表現を含んでいる文字列を返す。このメソッドは、ここでは関連するログメッセージを実行中に標準出力にプリントするために用いられている。SortMachine クラスに対する参照は、コード生成されたクラス CGBase で定義された `ObjGet_vdm_SortMachine` 関数を呼び出すことで得られる。

```
cout << "Evaluating DoSort(" << arr1.ascii () << "):\n";
type_ref_Sorter dos (ObjectRef (new vdm_DoSort ()));
res = ObjGet_vdm_SortMachine(smach)->
      vdm_SetAndSort (dos, arr1);
cout << res.ascii() << "\n\n";
```

ExplSort クラスで定義されたソートアルゴリズムで `arr2` をソートするため、同じような次のコードが書ける:

```
cout << "Evaluating ExplSort(" << arr2.ascii () << "):\n";
type_ref_Sorter expls (ObjectRef(new vdm_ExplSort ()));
res = ObjGet_vdm_SortMachine(smach)->
      vdm_SetAndSort (expls, arr2);
cout << res.ascii() << "\n\n";
```

ImplSort クラスで実装されたソートアルゴリズムで `arr2` をソートするため、次のコードが書ける:

```
cout << "Evaluating ImplSort(" << arr2.ascii () << "):\n";
type_ref_Sorter imps (ObjectRef(new vdm_ImplSort ()));
res = ObjGet_vdm_SortMachine(smach)->
      vdm_SetAndSort (imps, arr2);
cout << res.ascii() << "\n\n";
```

注意したいのは、コードに対するインターフェイスは陰や陽の関数 / 操作をもつことに依存しないということだ。

同様に、ImplSort に対して事後条件関数を呼び出したい場合を想定できるだろう。これに対してコードジェネレータは `vdm_ImplSort` クラス内に `vdm_post_ImplSorter` という名称の関数を生成した。これは通常の方法で呼び出すことができる。

```
cout << "Evaluating post condition for ImplSort:\n";
Bool p = ObjGet_vdm_ImplSort(imps)->
    vdm_post_ImplSorter (arr2, res);
cout << "post_ImplSort(" << arr2.ascii () << ", " <<
    res.ascii () << "):\n" << p.ascii () << "\n\n";
```

SortMachine クラスの SetAndSort メソッドを呼び出す代わりに、16行から18行目で示すように最初は SetSort を呼び出すことで、求めるソートアルゴリズムの設定を選択できる。ここでは MergeSort アルゴリズムを選択するが、結果の C++ コードに実行時エラーが含まれてしまうことが知られている。結果のコードを次に示す:

```
type_ref_Sorter mergs (ObjectRef(new vdm_MergeSort ()));
ObjGet_vdm_SortMachine(smach)->vdm_SetSort (mergs);

cout << "Evaluating MergeSort(" << arr2.ascii () << "):\n";
res = ObjGet_vdm_SortMachine(smach)->vdm_GoSorting(arr2);
cout << res.ascii() << "\n\n";
```

記述のメインプログラムは sort\_pp.cc という名称のファイルに実装され、これは付録 B.8 に一覧されている。

### 3.2.4 生成された C++ コードの部分的置き換え

最後に、生成された C++ コードを手書きコードへ置き換える可能性についていくつか述べておくべきだ。これが役に立つ状況として、主な2つを挙げることができる:

- コードジェネレータでサポートされない構成要素に対してコード実装を行いたい場合。
- 今ある構成要素をより効果的に実装したい場合。

ソート例題に関して、MergeSorter 関数の手書きコード版を実装したいという場合が想定されるが、そこにコードジェネレータでサポートされていない構成要

素が含まれているからである。そのためコード生成された関数 `vdm_MergeSort::vdm_MergeSorter` を手書き版に置き換えることが、ユーザーにとっては必要となる。そのためには新しい関数を書く必要がある。これは `MergeSort.cc` の生成コード版と同じ宣言ヘッダーをもたなければならない。

```
type_rL vdm_MergeSort::vdm_MergeSorter(const type_rL &vdm_l) {  
    ...  
}
```

生成コード関数を手書き関数と置き換えるために、この関数は `MergeSort_userimpl.cc` という名のファイルに実装される必要があり、さらに以下の2つの定義が `MergeSort_userdef.h` ファイルに追加される必要がある:

```
#define DEF_MergeSort_USERIMPL  
    // a user defined file is now included.Note: For classes  
    // containing implicit functions/operations, an user  
    // implemented file is a presumption and this line should be  
    // obmitted.  
#define DEF_MergeSort_MergeSorter  
    // the MergeSorter function in class MergeSort is handcoded
```

このようにコードジェネレータで生成された特定の関数は手書き関数に置き換えることができる。

### 3.3 C++コードのコンパイル、リンク、実行

ユーザーが前に述べたファイルの手書きを行った場合、その後はC++コードのコンパイル、リンク、実行を行うこととなる。

VDM++ to C++ コードジェネレータのこの版で生成されたC++コードは、以下のサポート対象コンパイラを用いてコンパイルされなければならない:

- 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

VDM++ to C++ コードジェネレータ のこの版で生成された C++ コードは、以下のサポート対象コンパイラを用いてコンパイルされなければならない:

- libCG.a: コード生成補助関数。このライブラリは VDM++ to C++ コードジェネレータ と共にリリースされたもので、付録 A に記載されている。
- libvdm.a: VDM C++ ライブラリ。このライブラリは VDM++ to C++ コードジェネレータ と共にリリースされたもので、[1] に記載されている。
- libm.a: コンパイラ対応の数学ライブラリ。

ソート例題の実装で用いられる Makefile は付録 C に一覧されている。メインプログラム sort\_pp をコンパイルするために、make sort\_pp とタイプする必要がある。

ここまでくれば、メインプログラム sort\_pp の実行が可能になる。結果は以下にリストされている。MergeSort の実行中に実行時エラーがおきてしまうことには注意しよう。これは、サポートされていない構成要素を実行しようとした場合に引起される。生成コードに含まれている位置情報が、基礎を成す仕様におけるエラーを導く。

```
$ sort_pp
Evaluating DoSort([ 3,5,2,23,1,42,98,31 ]):
[ 1,2,3,5,23,31,42,98 ]

Evaluating ExplSort([ 3,1,2 ]):
[ 1,2,3 ]

Evaluating ImplSort([ 3,1,2 ]):
[ 1,2,3 ]
```



```
Evaluating post condition for ImplSort:
post_ImplSort([ 3,1,2 ],[ 1,2,3 ]):
true

Evaluating MergeSort([ 3,1,2 ]):
Last recorded position:
In: MergeSort. At line: 26 column: 18
The construct is not supported: Sequence concatenation pattern
$
```

## 4 サポートされていない構成要素

この版のコードジェネレータ では以下の VDM++ 構成要素はサポートされていない:

- 式:
  - ラムダ式。
  - 合成式、繰り返し式、関数の同値式。
  - 関数型インスタンス化式。ただし以下の例題にあるように、コードジェネレータは適用式と組み合わせて関数型インスタンス化式をサポートする:

```
Test:() -> set of int
Test() ==
  ElemToSet[int](-1);

ElemToSet[@elem]: @elem +> set of @elem
ElemToSet(e) ==
  {e}
```

- VDM++の同時平行性の部分で、`#act`、`#fin #active`、`#waiting`、`#req`の式。

- 文:

- always 文、exit 文、 trap 文、 recursive trap 文。
- start 文と start list 文。

- 次における型束縛 ([3] 参照):

- let-be-st 式 / 文。
- 列、集合、写像の包含式。
- iota 式と量化式。

例題として次の式は コードジェネレータによってサポートされている:

```
let x in set numbers in x
```

一方、次はサポートされていない (原因は型束縛  $n: \text{nat}$ ):

```
let x: nat in x
```

- パターン:

- 集合和パターン。
- 列連結パターン。

- スレッド

- 同期定義

コードジェネレータ はこれらの構成要素を含む仕様に対してコンパイル可能なコード生成ができるが、サポートされない構成要素を含む部門が実行された場合、コードの実行が実行時エラーという結果になってしまう。以下の関数定義を考えてみよう:

```
f: nat -> nat
f(x) ==
  if x <> 2 then
    x
  else
    iota x : nat & x ** 2 = 4
```

この場合  $f$  に対するコードが生成されコンパイルされる。 $f$  に対応してコンパイルされた C++ コードは、 $f$  に値 2 が与えられた場合には、iota 式の型束縛がサポートされていないため、結果は実行時エラーとなる。

サポートされない構成要素に出会った場合は常に コードジェネレータ が警告を与えることに注意しよう。

## 5 VDM 仕様のコード生成 - 詳細

この章では、クラス、型、関数、操作、インスタンス変数、値、式、そして文、といった VDM++ 構成要素が、コード生成される方法を詳細に述べる。

コードジェネレータ を専門的に使用したい場合は、集中的にこの解説を学習するべきだろう。

注意: この章では様々な VDM++ 構成要素とその C++ コードへのマップに焦点を当てていて、生成された C++ ファイルの全体的構造については述べていない。読者には第 2.4 章と第 3 章が、全体的構造の記載の参照 となる。

### 5.1 生成コードクラス

VDM++ クラス各々に対して相当する C++ クラスが生成される。VDM++ クラスの継承構造は正確に、生成された C++ クラスの継承構造に反映される。しかし、コードジェネレータ で生成される型すべてにあてはまることだが、生成されたクラスは VDM C++ ライブラリと堅く対応して組となる。型システムが生成されたコード中でどのように動くかを完全に理解するためには、このライブラリ [1] の説明を読むべきであろう。それでも以下に少しだけ VDM C++ ライブラリ の紹介を行おう。

#### 5.1.1 VDM C++ ライブラリにおけるオブジェクト参照

VDM C++ ライブラリ は 2 つのスーパークラス: *Common* と *MetaivVal* で構成される。VDM++ の型すべてに対して、対応する *Common* のサブクラスの

C++ クラスが存在し、したがって VDM++ 中の全種類の値に対して相当する *MetaivVal* の C++ クラスが存在する。このように VDM C++ ライブラリ は型と値において構造化されたシステムであり、これについて 図 6 に図解されている。

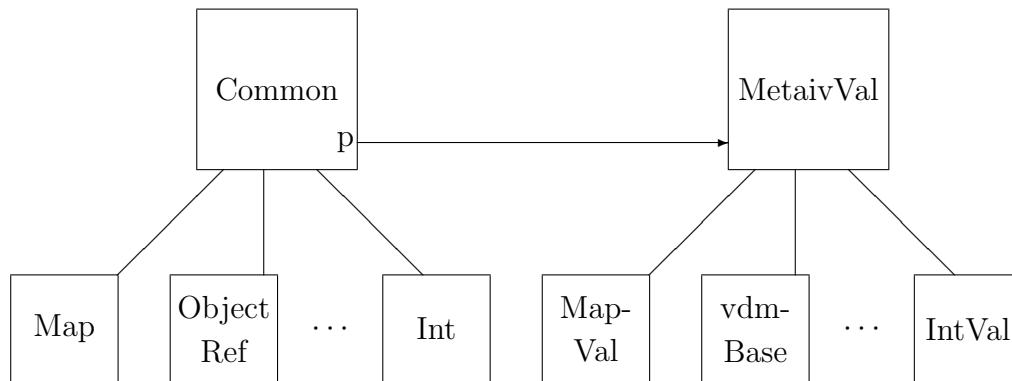


図 6: VDM C++ ライブラリ の全体構造

あるクラスにあるオブジェクトが生成され *Int* とすると、*IntVal* のオブジェクトもまた自動的に生成され、これが整数値を含む。加えて、*Int* オブジェクトからのポインター *p* は、*IntVal* オブジェクトを指すように設定される。

ここでどのように VDM++ の型である“オブジェクト参照”が VDM C++ ライブラリ中に反映されるのかを見てみよう。他のすべての型に対すると同様、VDM C++ ライブラリ は2つのクラス: 値部分システムにはクラス *vdmBase* を、型部分システムにはクラス *ObjectRef* を提供する。*ObjectRef* のインスタンス生成の場合、コンストラクタは相当する値部分側のオブジェクト、この場合クラス *vdmBase* のオブジェクト、に対するポインタを入力としてとる。*ObjectRef* クラスで用いられる、典型的なコンストラクタ宣言の一例を次に示す:

```
class ObjectRef : public Common {
public:
    ...
    ObjectRef(vdmBase* = NULL);
    ...
}
```

### 5.1.2 クラスの生成コードの継承構造

コードジェネレータは以下の方法で、VDM C++ ライブラリのオブジェクト参照サポートを利用する。VDM++ クラスに相当する全 C++ クラスは、vdmBase クラスから継承する。加えて、VDM++ 仕様の全クラスに対して、正確にその VDM++ クラスのオブジェクト参照を表す相当のクラスが生成される。この C++ クラスは ObjectRef クラスから継承する。

ソート例題で生成された C++ クラスと VDM C++ ライブラリの継承構造を、[図 7](#)で見ることができる。

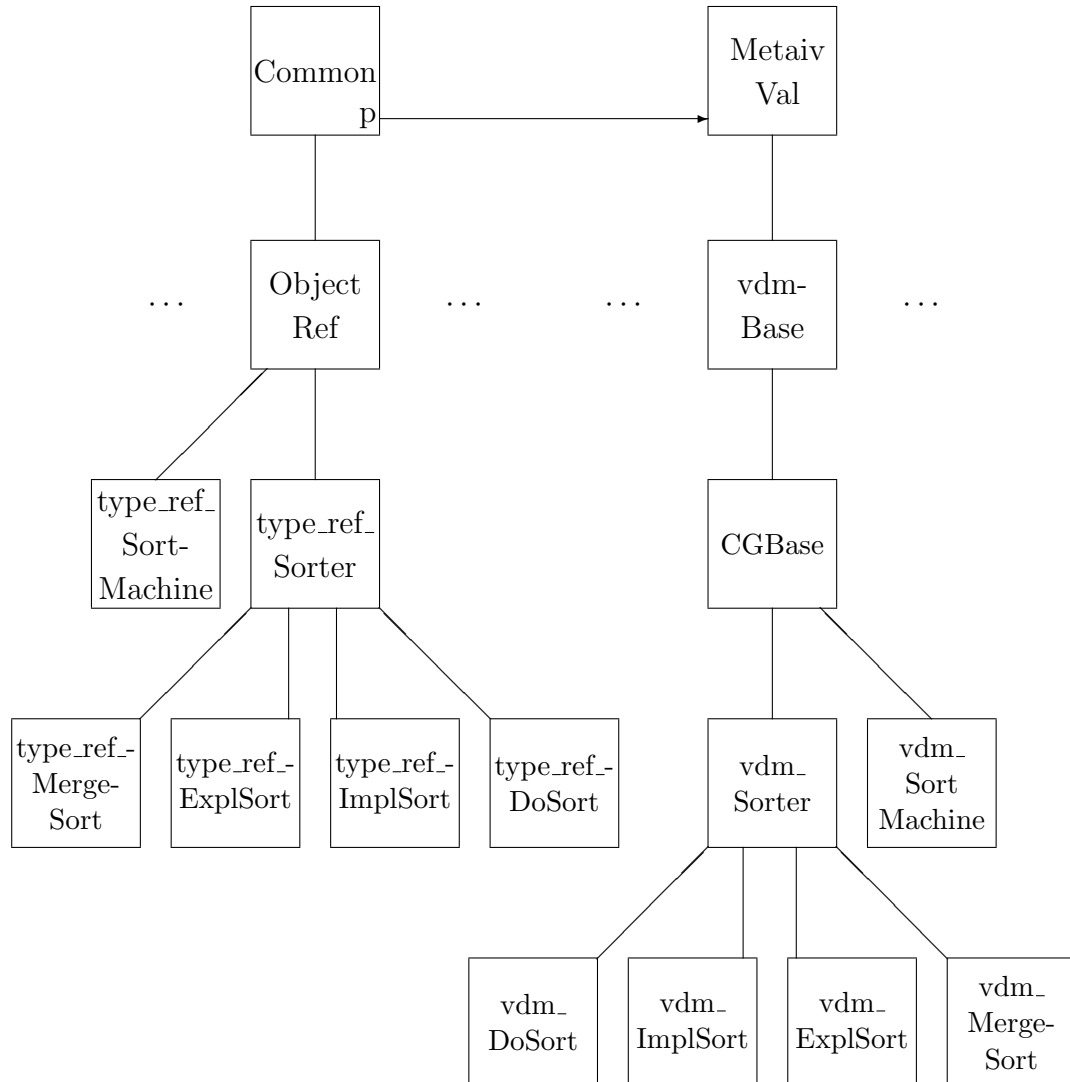


図 7: ソート例題の C++クラスと VDM C++ ライブラリ の継承構造

型システムにおいて部分特化したオブジェクト参照クラスは、各 VDM++ クラスに対して生成された `type_ref_<Classname>` である。宣言はクラス `type_ref_DoSort` と考えよう:

```
class type_ref_DoSort : public virtual type_ref_Sorter {
```

```
public:
    type_ref_DoSort() : ObjectRef() {}
    type_ref_DoSort(const Generic &c) : ObjectRef(c) {}
    type_ref_DoSort(vdmBase * p) : ObjectRef(p) {}
    const char * GetTypeName() const { return "type_ref_DoSort"; }
};
```

このクラスは、`vdmBase` クラスに対するポインターを取り入れるコンストラクタを含んでいる。クラス `ExplSort` のオブジェクトに対するオブジェクト参照構築は、次の方法で行うことができる:

```
type_ref_DoSort ds (new vdm_DoSort());
```

これは 図 7 で見るように、生成される C++ クラスは `vdmBase` から直接の継承はせず `CGBase` クラスを中継する。またこのクラスもコードジェネレータで生成される。

`CGBase` クラスは `CGBase.cc` と `CGBase.h` のファイル中で宣言と定義がなされる。クラス定義は別として、`CGBase` ファイルもまたいくつかの外部関数の定義からなる。それでも、このコードでオブジェクト参照型の値部分 (これが実際の C++ オブジェクト参照である) の取り出しができる関数を提供する。

*Sorting* 例題に対する `CGBase.h` ファイルの取り出しを考えよう:

```
class CGBase : public vdmBase {
private:
    ....
public:
    virtual vdm_DoSort * Get_vdm_DoSort() { return 0; }
    ....
    virtual vdm_Sorter * Get_vdm_Sorter() { return 0; }
};
vdm_DoSort * ObjGet_vdm_DoSort(const ObjectRef &obj);
...
vdm_Sorter * ObjGet_vdm_Sorter(const ObjectRef &obj);
enum {
```

```
VDM_DoSort,  
...  
VDM_Sorter  
};
```

各 VDM++ クラスに対しグローバル関数: `ObjGet_vdm_<ClassName>.` が生成される。この関数は *ObjectRef* をとり入れて、相当する C++ クラスオブジェクトに対するポインターを返す。さらに、クラス記述それぞれに単一タグが定義される。

DoSort クラス内にオブジェクト参照を構築し関数を適用する例題が、以下に与えられている:

```
type_iL somelist;  
type_ref_DoSort ds (new vdm_DoSort);  
ObjGet_vdm_Sorter(ds)->vdm_Sort(somelist);
```

VDM C++ ライブラリ の実装は参照カウンタに基づくので、`vdm_A` クラス実装に対するポインターはこれを参照する *ObjectRef* クラスオブジェクトが存在しない場合は削除される。これを示すために以下の例題を考えてみよう:

```
{  
    type_ref_DoSort ds(new vdm_DoSort);  
    {  
        type_ref_DoSort tmp(new vdm_DoSort);  
        ds = tmp; // at this point the first vdm_DoSort  
                  // pointer will be deleted.  
    }  
} // The second vdm_DoSort pointer will be deleted when  
  // this scope is closed.
```

`vdm` クラスに対し、ポインターを直接宣言しこれをインスタンス化して *ObjectRef* とすることは、決して行ってはいけない。これは VDM C++ ライブラリ が参照カウンタに基づくため誤った結果を導く可能性があり、オブジェクト参照を指すポインターが (少なくとも VDM C++ ライブラリの視点から見て) ある場合に、それらオブジェクトを削除してしまうからである。



```
{
    vdm_DoSort * ds_p = new vdm_DoSort();           // Never do this
    {                                               // Never do this
        type_ref_DoSort tmp(ds_p);                 // Never do this
        ...                                         // Never do this
    } // Now the ds_p will be deleted.             // Never do this
    ...                                           // Never do this
}
```

### 5.1.3 生成されたクラスの構造

生成された C++ クラスは次を含む:

- VDM++ の関数と操作を実装する C++ 関数。
- オブジェクト参照実践のための補助関数。
- クラスのコンストラクタ/デストラクタ。

クラスのアクセス修飾子は VDM++ クラスで指定されるものに続けるが、アクセス修飾子を与える意味がない型定義では除かれる。このように、たとえば VDM++ 段階でパブリックな関数は、相当する C++ クラスのパブリックな要素としてコード生成されることになる、といった具合である。

C++ vdm\_DoSort クラスの宣言を考えてみる:

```
class vdm_DoSort : public virtual vdm_Sorter {

    friend class init_DoSort ;

public:
    vdm_DoSort * Get_vdm_DoSort () { return this; }
    ObjectRef Self () { return ObjectRef(Get_vdm_DoSort()); }
    int vdm_GetId () { return VDM_DoSort; }
    vdm_DoSort ();
    virtual vdm_DoSort () {}
```

```
private:
    virtual type_iL vdm_DoSorting (const type_iL &);
    virtual type_iL vdm_InsertSorted (const Int &, const type_iL &);

public:
    virtual type_iL vdm_Sort (const type_iL &);
};
```

クラスは VDM++ 関数 `vdm_DoSorting`、`vdm_InsertSorted`、`vdm_Sort` になる。

補助関数は次の通り:

- `Get_vdm_DoSort`: オブジェクト自身に対する参照ポインタを返す。
- `Get_vdm_Self`: オブジェクト自身に対するオブジェクト参照ポインタを返す。
- `vdm_GetId`: クラスの単一タグを返す。

## 5.2 型のコード生成

第 3.1 章ですでに、VDM++ 型から C++ コードへのマップの方法について、簡単な導入を行った。

ここではもう少し詳しい説明を行う。

第 5.2.1 章では、VDM++ 型をコード生成するときに用いる方策に対して動機付けを行う。その後第 5.2.2 章で、各 VDM++ 型から C++ コードへのマップを説明する。第 5.2.3 章では、型に対して用いる名称仕様をまとめている。

### 5.2.1 動機付け

コードジェネレータの型スキームは2つの部分に分けることができる:

- 生成された C++ コードの関数ヘッダーに用いられる型スキーム。

- 生成された C++ コードの残りの部分に用いられる型スキーム。

関数ヘッダーに用いる型スキームは、コード生成された C++ の型を使用する。生成されたコードの残り部分に用いる型スキームは、VDM C++ ライブラリで見られる各 VDM++ データ型の固定実装を用いる。VDM C++ ライブラリ (`libvdm.a`) は [1] で記述する。

入力パラメータとして `seqofchar` をとる VDM 関数があると考えてみよう。相当する C++ 関数は、したがって入力として `type_cL` 型のパラメータをとる。型 `type_cL` はコード生成された型であり、この `c` は VDM 型 `char` に似ている。しかし関数実装においては型 `type_cL` の代わりに、唯一 VDM C++ ライブラリの型 `Sequence` を用いる。

コード生成される型は明らかに、生成された C++ コードをよりよいものとする。コンパイル時にさらなる型エラーを捉えることができ、ユーザーに対してより多くの情報提供を行うものとなる。

新しい型を導入することで新しい問題も発生する：つまり、クラス A における `char+` の `seq` は、クラス B における `char+` の `seq` と同じ型であることを、保証しなければならない。

```
class A
types
  C = seq of char
end A

class B
types
  D = seq of char
end B
```

VDM++ で、型 A'C と B'D は同等である。しかしこれらは C++ においては異なる、なぜなら C++ では基本データ型を別にすれば名称同値を用いるからである。

生成されたコードは2つのことを保証しなければならない:

- 匿名 VDM++ 型は、生成コード中の型の正当性を保証するために、コード生成が一度だけ許されている。

- 生成された型名称は読み取り可能、判別可能である必要がある。

最初の問題は、<Class Name>\_anonym<Class Name>\_anonym.h<Class Name>\_anonym.cc はこれらの型の実装を含む。さらに、それらのマクロ定義を含む。

匿名でない型つまり合成型と型名称は、<Class Name>\_anonym.h ファイルではなく、代わりに<Class Name>.h

前に挙げた例題中で、クラス A に対して生成された匿名ヘッダーファイルを見よう。

A\_anonym.h は次のようなものである：

```
class type_cL;

#define TYPE_A_D type_cL

#ifndef TAG_type_cL
#define TAG_type_cL (TAG_A + 1)
#endif

#ifndef DECL_type_cL
#define DECL_type_cL 1
class type_cL : public SEQ<Char>
...

#endif
```

最初の #define 文で、モジュール A 中の型 D に対してマクロを定義する。これは型 type\_cL で置き換えられる。TAG\_type\_cL は、生成されたコードの型 type\_cL に対する単一タグを保証する。2つの #ifndef 文が、ファイル A\_anonym.h か B\_anonym.h のいずれかで、TAG\_type\_cL と type\_cL が1度だけ定義されることを保証する。

第2の問題は、生成された C++ 型に対して選ばれた名称仕様によって解決される。型名称生成のために、型を規範形式と呼ばれるものに展開し、その後に規範形式に含まれる型名称と型コンストラクタに基づいて名称を与える。以下の2つの節でさらに用いた表記表についての情報が得られる。

### 5.2.2 VDM++ 型から C++ へのマップ

この章ではどのように VDM 型が C++ 型にマップされるかを述べる。

- **ブール型**

VDM `bool` 型は VDM C++ ライブラリクラス `Bool` にマップされ、文字 `b` で略される。

- **数値型**

VDM `nat`、`nat1`、`int` 型はすべて VDM C++ ライブラリクラス `Int` にマップされ、文字 `i` で略される。VDM `real` と `rat` 型は VDM C++ ライブラリクラス `Real` にマップされ、文字 `r` で省略される。

- *The Character Type* 文字型

VDM 型 `char` は VDM C++ ライブラリクラス `Char` にマップされ、文字 `c` で略される。

- *The Quote Type* 引用型

VDM 引用型は VDM C++ ライブラリクラス `Quote` にマップされ、文字 `Q` で略される。すべての型に対するのと同様に、引用型に対しても単一タグが保証されなければならない。以下でどのように引用 `<Hello>` がコード生成されるかを示す。

以下のコードがファイル `<ClassName>_anonym.h` に追加される:

```
extern const Quote quote_Hello;
#define TYPE_A_C Quote
#ifdef TAG_quote_Hello
#define TAG_quote_Hello (TAG_A + 1)
#endif
```

以下のコードがファイル `<Clasname>_anonym.cc` に加わる:

```
# if !DEF_quote_Hello && DECL_quote_Hello
# define DEF_quote_Hello 1
const Quote quote_Hello("Hello");
#endif
```

このように宣言された引用値は C++ コード中で `quote_Hello` として参照してよい。

- トークン型

トークン型は C++ クラス `Record` を用いて実装される。しかしトークンレコードのタグは常に `TOKEN` と等しく、これはファイル `cg_aux.h` (付録 A 参照) 内で宣言されるマクロで、トークンレコードの項目数は常に 1 である。VDM++ 値である `mk_token(<HELLO>)` は、たとえば 以下のように構成され得る:

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

- 列型

合成型である `set`、`sequence`、`map`、を扱うために、テンプレートが導入される。これらのテンプレートは VDM C++ ライブラリでも定義されるが、C++ VDM ライブラリ中の C++ クラスである `Set`、`Sequence`、`Map` を基としている。

`seq` 型は文字 `L` で略される。例としてどのように `int+` の VDM 型がコード生成されるのか見よう:

```
class type_iL : public SEQ<Int> {
public:
    type_iL() : SEQ<Int>() {}
    type_iL(const SEQ<Int> &c) : SEQ<Int>(c) {}
    type_iL(const Generic &c) : SEQ<Int>(c) {}
    const char * GetTypeName() const { return "type_iL"; }
};
```

VDM `seq` 型はテンプレートの `SEQ` クラスから継承するクラスにマップされる。`int+` の `seq` の場合、テンプレートクラスの引数は `\pathInt+` であり、VDM 基本型 `int` を表わす C++ クラスである。新しいクラス名称は次のように作成される:

```
type : 匿名型を示す
i: 整数を示す
L: 列を示す
```

type\_iL クラスに対するいくつかのコンストラクタが、GetTypeName 関数と共に生成されていることにもまた注意しよう。

- 集合型

VDM set 型は VDM seq 型と同じ方法で扱われる。SEQ よりむしろテンプレートクラス SET が用いられ、型は文字 S で略される。

- マップ型

VDM map 型は VDM seq 型と同じ方法で扱われる。SEQ よりむしろテンプレートクラス MAP が用いられ、Map テンプレートクラスは引数は 1 つではなく 2 つとる。map 型は文字 M で略される。

- 合成型/レコード型

各合成型は、VDM C++ ライブラリ Record クラスのサブクラスであるクラスにマップされる。たとえば、クラス M で定義された次の合成型

```
A:: c : real
    k : int
```

これは以下のようにコード生成される:

```
class TYPE_M_A : public Record {
public:
    TYPE_M_A() : Record(TAG_TYPE_M_A, 2) {}
    TYPE_M_A(const Generic &c) : Record(c) {}
    const char * GetTypeName() const { return "TYPE_M_A"; }
    TYPE_M_A &Init(Real p1, Int p2);

    Real get_c() const;
    void set_c(const Real &p);
    Int get_k() const;
    void set_k(const Int &p);
};
```

分かる通り、クラス M の名称 A のレコードには名称: TYPE\_M\_A が与えられる。

いくつかの要素関数が生成された C++ クラス定義に追加されている:

- 2つのコンストラクタが追加された。
- 関数 `GetTypeName` が追加された。
- 初期化関数 `Init` が追加されている。この関数は、入力パラメータの相当する値に対するレコード項目を初期化し、オブジェクトに対する参照を返す。
- レコード中の各項目に対して、その値を獲得したり設定したりするために、2つの要素関数が追加される。これら関数の名称は相当するVDMレコード項目切替の名称と一致する。項目切替がない場合、レコード中の要素位置、たとえば `get_1`、が代わりに用いられる。

`Init` 関数と `set/get` 関数の実装は、レコード型が定義されたクラスの実装ファイル中に見つけることができる。前述のレコード型に対して、ファイル `M.cc` 中に以下のコードを見つけることができる:

```
TYPE_M_A &TYPE_M_A::Init(Real p1, Int p2) {
    SetField(1, p1);
    SetField(2, p2);
    return * this;
}

Real TYPE_M_A::get_c() const { return (Real) GetField(1); }
void TYPE_M_A::set_c(const Real &p) { SetField(1, p); }
Int TYPE_M_A::get_k() const { return (Int) GetField(2); }
void TYPE_M_A::set_k(const Int &p) { SetField(2, p); }
```

#### ● 組型/直積型

組を扱う方策は合成型を扱う方法のものによく似ている。各組型はVDM C++ ライブラリ `Tuple` クラスのサブクラスにマップされる。たとえば次の組:

```
int * real
```

これは以下のようにコード生成される:

```
class type_ir2P : public Tuple {
public:
```



```
type_ir2P() : Tuple(2) {}  
type_ir2P(const Generic &c) : Tuple(c) {}  
const char * GetTypeName() const { return "type_ir2P"; }  
type_ir2P &Init(Int p1, Real p2);  
Int get_1() const;  
void set_1(const Int &p);  
Real get_2() const;  
void set_2(const Real &p);  
} ;
```

新しいクラス名称は次のように作成される:

type: 匿名型を示す  
i: 整数を示す  
r: 実数を示す  
2P: 2つの部分型 / 要素をもつ組を示す

ただし合成型と組型のコード生成で1つ違いがある。VDM++ 組型は匿名型である。そのため、C++ の型定義は<ClassName>\_anonym.h ファイルにあり <Classname>.h ファイルにはない。同様に、要素関数の実装は<Classname>\_anonym.cc ファイルにあり <Classname>.cc ファイルにはない。

- ユニオン型

ユニオン型は VDM C++ ライブラリ Generic クラスへマップされる。

- オプション型

オプション型は VDM C++ ライブラリ Generic クラスへマップされる。

注意したいのは、nil が特殊な VDM++ 値である (型ではない) ことだ。

- オブジェクト参照型

第 5.1 章で、どのように2つの C++ クラスが各々の VDM++ クラスに対して生成されるかを記述する。これらのうちの1つはそのクラス自身を表し、もう1つは VDM++ クラスに対する参照を扱うために用いられる。

次の例題を見よう:

```
class M
types
A = seq of N
end M

class N
...
end N
```

クラス M で、クラス N のオブジェクトに対する参照の型 A を定義する。この例題をコード生成するとき、5 つのクラス: vdm\_M、vdm\_N、type\_ref\_M、type\_ref\_N、type\_1NRL、が定義される。最後は定義型 seqofN を表す。新しいクラス名称は次のように作成される:

type: 匿名型を示唆する  
1: クラスの名称中の文字数を示唆する  
N: クラス N を示唆する N  
R: オブジェクト参照を示唆する  
L: 列を示唆する

さらに、次のマクロがファイル M\_anonym.h 中で定義される:

### 5.2.3 VDM++ 型名称のコード生成

VDM++ と C++ の型システムは、C++ が名前等価を用いるのに対して VDM++ が構造等価を用いるため異なる。VDM++ において次の場合

```
type
  A = seq of int;
  B = seq of int
```

型 A と B は構造的に等しいので同値である。しかし、C++ で相当する例題では A と B の名称は異なるため同値ではない。

コードジェネレータではこの問題を、構造的に等しい型に対して同じ名称を生成することによって解決する。このように、対応して生成される C++ コードは (実質的には) 次の通り:

```
class type_iL

class type_iL : public SEQ<Int> {
public:
    ...
} ;

#define TYPE_ClassName_A type_iL
#define TYPE_ClassName_B type_iL
```

このようにすべての型名称定義は、`#define` 命令を通して型定義の構造的内容を反映した名称に定義される。

生成された型名称は `TYPE` で前置され、`class name` で後置され、型が定義され最後に選択された VDM 名称が連結される。すべての匿名型、つまり VDM++ 仕様で名称の与えられていない型は、`type` と型構造を反映する構成名を前につける。型名称は、VDM 型の展開と逆向きに洗練された表記法の使用を基にしている。

+. The other VDM++ types can of course also be named and `+`, `Int`

以下のテーブルは名称付け仕様の概略を示す。VDM 型と型コンストラクタの名称は最初の列にある。2 行目は VDM 型に相当する名称の生成に対するスキームが一覧されている。2 列目で `<tp>'s` は相当する VDM 型に対して生成された型名称で置き換えるべきである。たとえば VDM 型 `map char to int` には名称 `ciM` が与えられるが、その理由は、`char` は `c` に変換され、`int` は `i` に変換され、マップ型コンストラクタはこの 2 引数の型を逆ポーランド記法演算子 `M` と結びつけて、`ciM` となるのである。名称仕様についてはさらに後で述べる。

VDM	変 換	使 用 例
bool	b	b
nat1	i	i
nat	i	i

VDM	変 換	使 用 例
int	i	i
real	r	r
rat	r	r
char	c	c
quote	Q	<Hello> は Q に変換される
token	T	token は T に変換される
set	<tp>S	set of char は cS に変換される
sequence	<tp>L	sequence of real は rL に変換される
map	<tp1><tp2>M	map set of int to char は iScM に変換される
product	<tp1>.. <td>int * char * sequence of real は icrS3P に変換される</td>	int * char * sequence of real は icrS3P に変換される
composite	<length><name>C	合成型 Comp は 4CompC に変換される。 <length> は合成型の名称に含まれる文字数であることに注意。
union	U	int   char   real は U に変換される
optional	<tp>0	[ int ] は i0 に変換される
object ref	<length><name>R	クラス C1 のオブジェクト参照は 2C1R に変換される。 <length> はオブジェクトクラスの名称に含まれる文字数であることに注意。
recursive type	F	T = map int to T として定義された型 T は iFM、これは最初の展開型であるが、に変換される。繰り返し型には常に、生成された型名称に含まれる名称 F が与えられる。この例外が繰り返し合成型で、上記の名称となる。詳細は以下の章を参照のこと

上のテーブルで、合成とオブジェクト参照を扱うために用いられる <length> 部分は、型名称の読み込みが曖昧にされない保証を行う。

型を展開し標準的に表わすことは、回帰型の存在により難しくなる。したがって回帰型の名称は名前 F で表現する。たとえば、A = map int to A における型 A

は型名称 `iFM` で表現する。型定義 `B = sequence of A` で `B` に対して生成される型は、`FL` となる。

合成型は展開されないで、名称で表わされる、たとえば `Comp :: ..` は型 `4CompC` で表現される。

#### 5.2.4 不変条件

仕様中で不変条件を型定義の制限に用いる場合は、不変条件関数も利用可能だ。不変条件関数は、これに関連した型定義 ([3] 参照) と同じスコープ内で呼び出すことができる。VDM++ to C++ コードジェネレータ は 不変条件に相当する C++ 関数定義を生成する。例として、次の `M` クラス内の VDM++ 型定義を考えてみよう:

```
S = set of int
inv s == s <> {}
```

VDM++ 関数 `inv_S` に相当する関数宣言は、以下に一覧されている。この宣言は、`M.h` ファイル中の C++ クラス `vdm_M` の保護部分におかれる

```
Bool vdm_inv_S(const type_iS &);
```

この不変条件の実装は、ファイル `M.cc` にある。

VDM++ to C++ コードジェネレータ は不変条件の動的チェックをサポートしないことに注意し、不変条件関数は明示的に呼び出さなければならない。

### 5.3 関数定義と操作定義のコード生成

VDM++において、関数と操作は明示的あるいは暗黙に定義することができる。VDM++ to C++ コードジェネレータ は、陰と陽の両関数および両操作定義の C++ 関数宣言を生成する。これらの宣言は `<ClassName>.h` ファイル内にある。

ここで生成された C++ 関数宣言の2つの例を見よう:

VDM++ クラス ExplSort 内の操作定義 Sort は明示的であり、結果として ExplSort.h ファイル内の vdm\_ExplSort クラスに次の C++ 関数宣言を導く:

```
virtual type_iL vdm_Sort(const type_iL &);
```

VDM++ クラス ImplSort 内の関数定義 ImplSorter は暗黙であり、結果として ImplSort.h ファイル内の vdm\_ImplSort クラスに次の C++ 関数宣言を導く:

```
virtual type_iL vdm_ImplSorter(const type_iL &);
```

注意: C++ 関数宣言は VDM++ の語義に相当するために仮想的にパブリックに宣言される。VDM++ においてすべての関数と操作は仮想的でかつパブリックであるが、C++ においてはそのように宣言されたときのみそうなる。

関数宣言の例をもう1つ示す。以下の VDM++ 仕様において関数 f を見よう。

```
class M
  types
    A :: ...;
    B = seq of int
  functions
    f: seq of int * B -> A
    ...
end M
```

次の関数ヘッダーが関数 f に対して生成される:

```
type_1NRL
vdm_M_f(type_iL p1, TYPE_M_B p2) {
  ...
}
```

関数シグニチャにおいては、型名称がむしろ相当する展開型の名称よりも用いられることに注意しよう。これが、できる限り VDM 仕様に近いものとなる。変数宣言においても同様の方策が用いられる。

## 明示的な関数および操作の定義

VDM++ to C++ コードジェネレータは、明示的な VDM++関数と操作定義に対する C++ 関数定義を生成する。これらの関数定義は相当する <ClassName>.cc ファイルに置かれる。前述の VDM++ クラス ExplSort における陽操作 Sort に対し、次の C++ 関数定義がファイル ExplSort.cc に追加される:

```
type_iL vdm_ExplSort::vdm_Sort(const type_iL &vdm_l) {  
    ...  
}
```

提示した例題では、どのように関数や操作の名称が生成されるかという考え方を与える: VDM 仕様におけるクラス M 内の関数または操作の名称 f は、名称: vdm\_M::vdm\_f となる。

## 陰関数定義と陰操作定義

明らかに陰関数と陰操作の定義に対しては、生成された C++ コードにいかなる C++関数定義も加えられない。そのかわり、実装ファイル中にコードジェネレータ インクルードプリプロセッサを生成する。陰関数を含めるクラス ImplSort に対し、以下のプリプロセッサが実装ファイル ImplSort.cc 内に現われる:

```
#include "ImplSort_userimpl.cc"
```

そしてファイル ImplSort\_userimpl.cc 内のクラス ImplSort の陰関数実装に対して、責任はユーザーにある。このファイルが生成されなければコンパイル時にエラーが起きること、また陰関数が実装されなければリンカーがメッセージを出すこと、に注意したい。陰関数と陰操作の実装についての情報は第 [3.2.2](#) 章を参照のこと。

## 事前条件と事後条件

操作仕様上の事後条件は VDM++ to C++ コードジェネレータによって無視される。しかし、関数に対して事前条件と事後条件が指定された場合、相当する事前関

数と事後関数が利用できる ([3] を参照)。これらの関数の各々に対して、C++関数宣言と C++関数定義が生成される。事前関数と事後関数は、相当する VDM++関数から得たアクセス修飾子をもつ、C++ 生成クラスの要素である。

クラス ImplSort 内で、関数 ImplSorter の事後条件に対して生成された C++コードを見よう:

以下の関数宣言は、ファイル ImplSort.h にある:

```
Bool vdm_post_ImplSorter(const type_iL &, const type_iL &);
```

この関数の実装はファイル ImplSort.cc にある。

事前条件と事後条件のランタイムチェックをオプションで生成ができる (グラフィカルユーザーインターフェイスにおいて相当するチェックボックスを選択することで、あるいはコマンドライン上で -P オプションを指定することで)。たとえば、ExplSort クラスの関数 RestSeq を考えよう: ExplSort class:

```
RestSeq: seq of int * nat -> seq of int
RestSeq(l,i) ==
  [l(j) | j in set (inds l \ {i})]
pre i in set inds l
post elems RESULT subset elems l and
  len RESULT = len l - 1;
```

事前条件と事後条件のランタイムチェックを共に行う場合、次のコードが生成される:

```
type_iL vdm_ExplSort::vdm_RestSeq (const type_iL &vdm_l,
                                   const Int &vdm_i) {
  if (!this->vdm_pre_RestSeq((Generic) vdm_l,
                             (Generic) vdm_i).GetValue())
    RunTime("Run-Time Error: Precondition failure in RestSeq");
  Sequence varRes_4;
  ...
  if (!this->vdm_post_RestSeq((Generic) vdm_l, (Generic) vdm_i,
                             (Generic) varRes_4).GetValue())
```



```
    RunTime("Run-Time Error: Postcondition failure in RestSeq");  
    return (Generic) varRes_4;  
}
```

この方法で、VDM++ 段階の表明は生成されたコードで評価できる。

生成された C++ 関数を手書き関数コードで置き換える

生成された C++ 関数を手書きの C++ コードと置き換えることができることは、述べておくべきだろう。第 3.2.4 章で、これが役に立つ場面と、手書きコードを生成コードにつなぐためにユーザーが踏むべき手順について、述べる。

## 5.4 インスタンス変数のコード生成

インスタンス変数のコード生成はたいへん直接的である。インスタンス変数は相当する C++ クラスの要素変数に変換される。これらの要素変数は、生成されたクラス定義の保護部分にある。

以下の VDM++ 内のインスタンス変数宣言を考えよう:

```
class A  
instance variables  
    public i: nat;  
    private j : real;  
    protected k: int := 4;  
    message: seq of char :=[];  
    inv len message <= 30;  
    ...  
end A
```

ファイル A.h 内で コードジェネレータ によって生成された相当する要素宣言は次のようになる:

```
class vdm_A : public virtual CGBase {  
  
private:  
    Real vdm_j;  
    Sequence vdm_message;  
protected:  
    Int vdm_k;  
public:  
    Int vdm_i;  
  
public  
    ...  
    vdm_A (); // constructor of class A  
};
```

クラス A に対するコンストラクタ関数の実装は、A.cc ファイル中で見ることができる。これは次に挙げたインスタンス変数を初期化する:

```
vdm_A::vdm_A() {  
    vdm_k = (Int) 4;  
    vdm_message = Sequence();  
    ...  
}
```

注意: インスタンス変数内で指定された不変条件定義は、コードジェネレータによって無視される。

さらに、型 `type_cL` の代わりになぜコードジェネレータは型 `Sequence` を生成するのか、たぶん疑問をもつことだろう。第 5.2 章ですでに述べたように、コード生成された型はユーザーに対するインターフェイス上でのみ用いられる。しかし内部的な使用のためには、コードジェネレータは VDM C++ ライブラリにある各 VDM++ データ型の固定実装を用いる。

## 5.5 値定義のコード生成

ここでは定数値の定義のために、生成されたコードの説明を行う。

VDM++ 値の定義は、生成された C++ クラスの静的な要素変数に変換される。値変数の初期化は、生成され “.cc” ファイル内で初期化された C++ `Init_<ClassName>` クラスでなされる。

以下の例題を考えてみよう:

```
class A
values
public mk_(a,b) = mk_(3,6);
c : char = 'a';
protected d = a + 1;
end A
```

生成されたヘッダーファイル `A.h` は次のようになる:

```
class vdm_A : public virtual CGBase {
private:
    static Char vdm_c;
protected:
    static Int vdm_d;
public:
    static Int vdm_a;
    static Int vdm_b;
    ...
end vdm_A
```

実装ファイルは次のようになる:

```
Char vdm_A::vdm_a;
Int vdm_A::vdm_b;
Int vdm_A::vdm_c;
```

```
Int vdm_A::vdm_d;

class init_A {
public:
    // constructor
    init_A() {
        ...
        ... pattern match code for the tuple pattern.
        vdm_A::vdm_a = (Int) 3;
        vdm_B::vdm_b = (Int) 6;

        vdm_A::vdm_c = (Char) 'a';
        vdm_A::vdm_d = vdm_A::vdm_a + (Int) 1
    }
}

// instantiation of class init_A
init_A Init_A;
```

## 5.6 式と文のコード生成

VDM++ 式と文は、生成コードが仕様の期待通り動作するようにコード生成される。

未定義式とエラー文は、関数 `RunTime` (付録 A 参照) の呼び出しに変換される。この呼び出しは実行を終了させ、未定義式が実行されたことの記録を行う。

## 5.7 名称仕様

仕様中の変数は、生成された C++コードにおける変数に変換される。VDM++ to C++ コードジェネレータ が用いる名称仕様は、全変数を: `vdm_<name>`、ここで `<name>` は仕様に表される名称、のように改名することである。関数 `f` はたとえば `vdm_f` と名づけられる。さらに以下の名称をコードジェネレータで用いる:

- `length_record`: レコード `record` 中の項目数を定義する静的変数。
- `pos_record_field`: レコード `record` 中の項目切替選択 `field` の 位置/添え字 (整数) を定義する静的変数。
- `name_number`: 生成 C++コードが使用する一時的変数。仕様/接続形態文はメソッド内で定義された順に 1 から番号付けされる。

仕様中の変数内に現れる下線 (‘\_’) やシングルクォート (‘’) は、下線 u (‘\_u’) と下線 q (‘\_q’) にそれぞれが各々交換される。

## 5.8 標準ライブラリ

### Math ライブラリ

Math ライブラリ (`math.vpp` ファイル) を用いた仕様がコード生成される場合、これらの関数は暗黙に定義されるため、ライブラリ機能が `MATH_userimpl.cc` という名のファイルに実装されなければならない。このファイルの既定実装は、ディレクトリ `vpphome/cg/include` にある。

### IO ライブラリ

IO ライブラリ (the `io.vpp` file) を用いた仕様がコード生成される場合、これらの関数は暗黙に定義されているため、ライブラリの機能は `IO_userimpl.cc` という名のファイルに実装されなければならない。このファイルの既定の実装は、ディレクトリ `vpphome/cg/include` にある。

IO ライブラリの `freadval` 関数からなる利用では、クラス 初期化関数が拡張される (初期化関数の詳細については第 5.5 章を参照)。`freadval` は、ファイルから VDM 値を読み込むために用いられる。レコード値を含むファイル上で正確に動作するために、関数 `AddRecordTag` は、テキストのタグ名称や生成コード内で用いられる整数タグ値の間に、正しい関係を設定するための初期化関数で提供される。`AddRecordTag` は `libCG.a` ライブラリ (付録 A 参照) の一部として提供されている。たとえば、クラス `M` がレコード型 `A` を定義すると仮定する。すると関数 `init_M` に次の行が現れる:

```
AddRecordTag("M'A", TAG_TYPE_M_A);
```

この方法で、型 M'A の値がファイルから読まれるとき、正しいタグをもつレコード値に変換されるはずである。

## 参考文献

- [1] CSK. *The VDM C++ Library*. CSK.
- [2] CSK. *VDM++ Installation Guide*. CSK.
- [3] CSK. *The VDM++ Language*. CSK.
- [4] CSK. *VDM++ Sorting Algorithms*. CSK.
- [5] CSK. *VDM++ Toolbox User Manual*. CSK.

## A libCG.a ライブラリ

ライブラリ libCG.a は、生成されたコードから利用される固定定義のライブラリである。libCG.a に対するインターフェイスは、cg.h と cg\_aux.h 中で定義される。

### A.1 cg.h

関数 RunTime と NotSupported は、ランタイムエラーが起きた時あるいはサポートされていない構成を含む分岐が実行された時に、呼び出される。これら両関数とも、エラーメッセージを印刷してプログラムは終了する (exit(1))。この関数のうちの 1 つが呼び出されたときに位置情報が利用できれば、VDM++ ソース仕様中で最後に記録された位置がプリントされる。これは、ランタイム位置情報オプションを用いてコード生成された場合である。

関数 `PushPosInfo`、`PopPosInfo`、`PushFile`、`PopFile`、が位置情報スタックの修正のために生成されたコードで用いられる。(ランタイム位置情報が生成コードに含まれる場合)。

`ParseVDMValue` は、IO 標準ライブラリの手動実装により用いられることが意図されている。ファイル名称と `Generic` 参照をとり込み、与えられたファイルから VDM 値を読む。この値は与えられた参照中にある。この関数は、成功したか否かにしたがって真または偽を返す。

```
/**
 * * WHAT
 * *   Code generator auxiliary functions
 * * ID
 * *   $Id: cg.h,v 1.16 2005/05/27 00:21:34 vdmtools Exp $
 * * PROJECT
 * *   Toolbox
 * * COPYRIGHT
 * *   (C) 2005, CSK
 ***/

#ifndef _cg_h
#define _cg_h

#include <string>
#include "metaiv.h"

void PrintPosition();
void RunTime(wstring);
void NotSupported(wstring);
void PushPosInfo(int, int);
void PopPosInfo();
void PushFile(wstring);
void PopFile();
void AddRecordTag(const wstring&, const int&);
bool ParseVDMValue(const wstring& filename, Generic& res);

// OPTIONS
```

```
bool cg_OptionGenValues();
bool cg_OptionGenFctOps();
bool cg_OptionGenTpInv();
#endif
```

## A.2 cg\_aux.h

cg\_aux.h 内の定義は、VDM++データ型<sup>4</sup>の実装に用いるライブラリから独立した補助的な定義を含める。

関数 Permute、Sort、Sortnls、Sortseq、IsInteger、GenAllComb、は式の様々な型に対応して生成されたコードで用いられる。

```
/**
 * * WHAT
 * *   Code generator auxiliary functions which are
 * *   dependent of the VDM C++ Library (libvdm.a)
 * * ID
 * *   $Id: cg_aux.h,v 1.14 2005/05/27 00:21:34 vdmtools Exp $
 * * PROJECT
 * *   Toolbox
 * * COPYRIGHT
 * *   (C) 2005, CSK
 */

#ifndef _cg_aux_h
#define _cg_aux_h

#include <math.h>
#include "metaiv.h"

#define TOKEN -3
```

---

<sup>4</sup>この版の VDM++ to C++ コードジェネレータでは VDM C++ ライブラリの使用のみが可能である。



```
Set Permute(const Sequence&);
Sequence Sort(const Set&);
bool IsInteger(const Generic&);
Set GenAllComb(const Sequence&);

#endif
```

## B C++ 手書きファイル

### B.1 DoSort\_userdef.h

```
#define TAG_DoSort 200
```

### B.2 ExplSort\_userdef.h

```
#define TAG_ExplSort 400
```

### B.3 ImplSort\_userdef.h

```
#define TAG_ImplSort 300
```

### B.4 MergeSort\_userdef.h

```
#define TAG_MergeSort 100
```

## B.5 SortMachine\_userdef.h

```
#define TAG_SortMachine 4500
```

## B.6 Sorter\_userdef.h

```
#define TAG_Sorter 4600
```

## B.7 ImplSort\_userimpl.cc

MergeSort の手書き版として `vdm_ImplSort::vdm_ImplSorter` の実装が選択された。

```
static type_iL Merge(const type_iL&, const type_iL&);

type_iL vdm_ImplSort::vdm_ImplSorter(const type_iL& l) {
    int len = l.Length();
    if (len <= 1)
        return l;
    else {
        int l2 = len/2;
        type_iL l_l, l_r;
        int i=1;
        for (; i<=l2; i++)
            l_l.ImpAppend(l[i]);
        for (; i<=len; i++)
            l_r.ImpAppend(l[i]);
        return Merge(vdm_ImplSorter(l_l), vdm_ImplSorter(l_r));
    }
}

type_iL Merge(const type_iL& _l1, const type_iL& _l2)
```

```
{
  type_iL l1(_l1), l2(_l2);
  if (l1.IsEmpty())
    return l2;
  else if (l2.IsEmpty())
    return l1;
  else {
    type_iL res;
    Real e1 = l1.Hd();
    Real e2 = l2.Hd();
    if (e1 <= e2)
      return res.ImpAppend(e1).ImpConc(Merge(l1.ImpTl(), l2));
    else
      return res.ImpAppend(e2).ImpConc(Merge(l1, l2.ImpTl()));
  }
}
```

## B.8 sort\_pp.cc

```
/**
 * * WHAT
 * *   Main C++ program for the VDM++ sort example
 * * ID
 * *   $Id: sort_pp.cc,v 1.14 2005/05/27 07:46:33 vdmtools Exp $
 * * PROJECT
 * *   Toolbox
 * * COPYRIGHT
 * *   (C) 2016 Kyushu University
 ***/

#include <fstream>
#include "metaiv.h"
#include "SortMachine.h"
#include "Sorter.h"
#include "ExplSort.h"
```

```
#include "ImplSort.h"
#include "DoSort.h"
#include "MergeSort.h"

// The main program.

int main(int argc, const char *argv[])
{
    // let arr1 = [3,5,2,23,1,42,98,31],
    //      arr2 = [3,1,2]:
    type_iL arr1 (mk_sequence(Int(3), Int(5), Int(2), Int(23), Int(1), Int(42), Int(98), Int(31)));
    type_iL arr2 (mk_sequence(Int(3), Int(1), Int(2)));

    // dcl smach : SortMachine := new SortMachine(),
    //      res : seq of int = [];
    type_ref_SortMachine smach (ObjectRef (new vdm_SortMachine ()));
    type_iL res;

    //      def dos : Sorter := new DoSort() in
    //      res = smach.SetAndSort(dos,arr1);

    wcout << L"Evaluating DoSort(" << arr1.ascii () << L"):" << endl;
    type_ref_Sorter dos (ObjectRef (new vdm_DoSort ()));
    res = ObjGet_vdm_SortMachine(smach)->vdm_SetAndSort (dos,arr1);
    wcout << res.ascii() << endl << endl;

    //      def expls : Sorter := new ExplSort() in
    //      res = smach.SetAndSort(expls,arr2);
    wcout << L"Evaluating ExplSort(" << arr2.ascii () << L"):" << endl;
    type_ref_Sorter expls (ObjectRef(new vdm_ExplSort ()));
    res = ObjGet_vdm_SortMachine(smach)->vdm_SetAndSort (expls,arr2);
    wcout << res.ascii() << endl << endl;

    //      def imps : Sorter := new ImplSort() in
    //      (res = smach.SetAndSort(imps,arr2)
    //      imps.Post_ImplSorter(arr2,res))
```

```
wcout << L"Evaluating ImplSort(" << arr2.ascii () << L"):" << endl;
type_ref_Sorter impls (ObjectRef(new vdm_ImplSort ()));
res = ObjGet_vdm_SortMachine(smach)->vdm_SetAndSort (impls,arr2);
wcout << res.ascii() << endl << endl;

wcout << L"Evaluating post condition for ImplSort:" << endl;
Bool p = ObjGet_vdm_ImplSort(impls)->vdm_post_ImplSorter (arr2, res);
wcout << L"post_ImplSort(" << arr2.ascii () << L"," << res.ascii () << L"):" <<
wcout << p.ascii () << endl << endl;

//      def mergs : Sorter := new MergeSort() in
//      smach.SetSort(mergs);
type_ref_Sorter mergs (ObjectRef(new vdm_MergeSort ()));
ObjGet_vdm_SortMachine(smach)->vdm_SetSort (mergs);

//      res = smach.GoSorting(arr2);
wcout << L"Evaluating MergeSort(" << arr2.ascii () << L"):" << endl;
res = ObjGet_vdm_SortMachine(smach)->vdm_GoSorting(arr2);
wcout << res.ascii() << endl << endl;

return 0;
}
```

## C Make ファイル

### C.1 Unix プラットフォームのための Make ファイル

```
# WHAT
#   Makefile for the code generated VDM++ sort example.
# ID
#   $Id: Makefile,v 1.24 2005/12/21 06:41:45 vdmtools Exp $
# PROJECT
#   Toolbox
# COPYRIGHT
#   (C) 2016 Kyushu University
#
# REMEMBER to change the variable TBDIR to fit your directory structure.
#

OSTYPE=$(shell uname)

GCC      = $(CC)
CXX      = $(CCC)
TBDIR    = ../../..
VPPDE    = $(TBDIR)/bin/vppde

INCL     = -I$(TBDIR)/cg/include

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
CCC     = $(CCPATH)clang++
else
ifeq ($(strip $(OSMV)),11) # 10.7
CC      = $(CCPATH)clang
CCC     = $(CCPATH)clang++
```

```
else
ifeq ($(strip $(OSMV)),10) # 10.6
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
else
ifeq ($(strip $(OSMV)),9) # 10.5
CC      = $(CCPATH)gcc-4.2
CCC     = $(CCPATH)g++-4.2
else
ifeq ($(strip $(OSMV)),8) # 10.4
CC      = $(CCPATH)gcc-4.0
CCC     = $(CCPATH)g++-4.0
else
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
endif
endif
endif
endif
endif
LIB      = -L$(TBDIR)/cg/lib -lCG -lvdm -lm -liconv
endif

ifeq ($(strip $(OSTYPE)),Linux)
CCPATH = /usr/bin/
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
LIB      = -L$(TBDIR)/cg/lib -lCG -lvdm -lm
endif

ifeq ($(strip $(OSTYPE)),SunOS)
CCPATH = /usr/sfw/bin/
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
LIB      = -L$(TBDIR)/cg/lib -L../lib -lCG -lvdm -lm
endif
```



```
ifeq ($(strip $(OSTYPE)),FreeBSD)
CCPATH = /usr/bin/
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
LIB     = -L$(TBDIR)/cg/lib -lCG -lvdm -lm -L/usr/local/lib -liconv
endif

CFLAGS = -g $(INCL)
CCFLAGS = $(CFLAGS)
CXXFLAGS= $(CCFLAGS)

OSTYPE2=$(word 1, $(subst _, ,$(strip $(OSTYPE))))
ifeq ($(strip $(OSTYPE2)),CYGWIN)
all: sort_pp.exe SortMain
else
all: sort_pp SortMain
endif

sort_pp.exe:
make -f Makefile.winnt

ALLFILES = DoSort ExplSort ImplSort MergeSort SortMachine Sorter

GENCCFILES = DoSort.cc DoSort.h DoSort_anonym.cc DoSort_anonym.h \
              ExplSort.cc ExplSort.h ExplSort_anonym.cc ExplSort_anonym.h \
              ImplSort.cc ImplSort.h ImplSort_anonym.cc ImplSort_anonym.h \
              MergeSort.cc MergeSort.h MergeSort_anonym.cc MergeSort_anonym.h \
              Sorter.cc Sorter.h Sorter_anonym.cc Sorter_anonym.h \
              SortMachine.cc SortMachine.h SortMachine_anonym.cc SortMachine_anonym.h \
              CGBase.cc CGBase.h

GENJAVAFILES = $(ALLFILES:%=%.java)

sort_pp : sort_pp.o $(ALLFILES:%=%.o) CGBase.o
$(CCC) -o sort_pp sort_pp.o $(ALLFILES:%=%.o) CGBase.o $(LIB)
```



```
DoSort.o: DoSort.cc DoSort.h DoSort_anonym.h DoSort_anonym.cc \
    CGBase.h Sorter.h DoSort_userdef.h
ExplSort.o: ExplSort.cc ExplSort.h ExplSort_anonym.h \
    ExplSort_anonym.cc CGBase.h Sorter.h ExplSort_userdef.h
ImplSort.o: ImplSort.cc ImplSort.h ImplSort_anonym.h \
    ImplSort_anonym.cc CGBase.h Sorter.h ImplSort_userimpl.cc \
    ImplSort_userdef.h
MergeSort.o: MergeSort.cc MergeSort.h MergeSort_anonym.h \
    MergeSort_anonym.cc CGBase.h Sorter.h MergeSort_userdef.h
SortMachine.o: SortMachine.cc SortMachine.h CGBase.h Sorter.h \
    MergeSort.h
Sorter.o: Sorter.cc Sorter.h CGBase.h
CGBase.o: CGBase.cc CGBase.h
sort_pp.o: $(GENCCFILES)
SortMain.class: SortMain.java $(GENJAVAFILES)
SortMain: SortMain.class $(GENJAVAFILES:%.java=%.class)

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

SPECFILES = dosort.vpp explsort.vpp implsort.vpp mergesort.vpp \
    sorter.vpp sortmachine.vpp

$(GENCCFILES): $(SPECFILES)
$(VPPDE) -c -P $^
```



```
$(GENJAVAFILES): $(SPECFILES)
$(VPPDE) -j -P $^

#####
#### Generation of postscript of the sort.tex document ####
#####

VDMLOOP = vdmloop

GENFILES = sort.aux sort.log sort.ind sort.idx sort.ilg vdm.tc

init:
cp mergesort.init mergesort.vpp

vdm.tc:
cd test; $(VDMLOOP)
cp -f test/$@ .

%.tex: $(SPECFILES) vdm.tc
vppde -lrNn $(SPECFILES)

sort.ps: $(SPECFILES).tex
latex sort.tex
makeindex sort
latex sort.tex
latex sort.tex
dvips sort.dvi -o

clean:
rm -f *.o sort_pp
rm -f *.class *.java.bak
rm -f sort.ps sort.dvi
rm -f $(SPECFILES:%=%.tex)
rm -f $(SPECFILES:%=%.aux)
rm -f $(GENFILES)
```

```
rm -f $(GENCCFILES)
rm -f $(GENJAVAFILES)
rm -f *.obj *.cpp *.exe
```

## C.2 Windows プラットフォームのための Make ファイル

```
##-----
##                               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
LINK     = link.exe

CFLAGS   = /nologo /c /MD /W0 /EHsc /D "WIN32" /TP

INCPATH  = -I"$(WTBDIR)/cg/include"

LDLAGS   = /nologo "$(WTBDIR)/cg/lib/CG.lib" "$(WTBDIR)/cg/lib/vdm.lib" user32.lib
```



## CG Version Files

```
CGSOURCES = DoSort.cpp DoSort.h DoSort_anonym.cpp DoSort_anonym.h \  
            ExplSort.cpp ExplSort.h ExplSort_anonym.cpp ExplSort_anonym.h \  
            ImplSort.cpp ImplSort.h ImplSort_anonym.cpp ImplSort_anonym.h \  
            MergeSort.cpp MergeSort.h MergeSort_anonym.cpp \  
            MergeSort_anonym.h Sorter.cpp Sorter.h Sorter_anonym.cpp \  
            Sorter_anonym.h SortMachine.cpp SortMachine.h \  
            SortMachine_anonym.cpp SortMachine_anonym.h CGBase.cpp CGBase.h
```

```
CGOBSJS = DoSort.obj ExplSort.obj ImplSort.obj MergeSort.obj \  
          SortMachine.obj Sorter.obj CGBase.obj
```

```
CGFLAGS = /D CG #-DDEBUG
```

```
sort_pp.exe: sort_pp.obj $(CGOBSJS)
```

```
sort_pp.obj: sort_pp.cpp SortMachine.h Sorter.h ExplSort.h \  
            ImplSort.h DoSort.h MergeSort.h  
DoSort.obj: DoSort.cpp DoSort.h DoSort_anonym.h DoSort_anonym.cpp \  
            CGBase.h Sorter.h DoSort_userdef.h  
ExplSort.obj: ExplSort.cpp ExplSort.h ExplSort_anonym.h \  
            ExplSort_anonym.cpp CGBase.h Sorter.h ExplSort_userdef.h  
ImplSort.obj: ImplSort.cpp ImplSort.h ImplSort_anonym.h \  
            ImplSort_anonym.cpp CGBase.h Sorter.h ImplSort_userdef.h \  
            ImplSort_userimpl.cpp  
MergeSort.obj: MergeSort.cpp MergeSort.h MergeSort_anonym.h \  
            MergeSort_anonym.cpp CGBase.h Sorter.h MergeSort_userdef.h  
SortMachine.obj: SortMachine.cpp SortMachine.h \  
                SortMachine_anonym.h SortMachine_anonym.cpp CGBase.h \  
                Sorter.h MergeSort.h SortMachine_userdef.h  
Sorter.obj: Sorter.cpp Sorter.h CGBase.h  
CGBase.obj: CGBase.cpp CGBase.h
```

```
SPECFILES = dosort.rtf explsort.rtf implsort.rtf mergesort.rtf \  
            sorter.rtf sortmachine.rtf
```

```
$(CGSOURCES): $(SPECFILES)
$(VPPDE) -c -P $^

all: sort_pp.exe

## Rules

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

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

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

%_userdef.h:
touch $@

#####
#### Generation of test coverage of the sort.tex document ####
#####

VDMLOOP = vdmloop

GENFILES = dosort.rtf.rtf explsort.rtf.rtf implsort.rtf.rtf \
          mergesort.rtf.rtf sorter.rtf.rtf sortmachine.rtf.rtf \
          vdm.tc

init:
cp mergesort.init mergesort.rtf

vdm.tc:
cd test; $(VDMLOOP)
cp -f test/$@ .
```



```
%.rtf.rtf: $(SPECFILES) vdm.tc
$(VPPDE) -lrNn $(SPECFILES)

clean:
rm -f $(CGOBS) sort_pp.obj cgexe.exe
rm -f *.cpp
rm -f $(CGSOURCES)
rm -f bigint_dl.lib bigint_dl.exp bigint_dl.pdb
```