

# VDMTools

---

C++コード生成マニュアル  
(VDM-SL)  
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-SL) 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

<b>1</b>	<b>導入</b>	<b>1</b>
<b>2</b>	<b>コードジェネレータの起動</b>	<b>2</b>
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	5
2.4	C++ 生成ファイル	6
<b>3</b>	<b>生成コードとのインターフェイス接続</b>	<b>7</b>
3.1	VDM-SL 型のコード生成 - 基本	7
3.2	ユーザーにより実装されたファイル	11
3.2.1	レコードタグに用いるオフセットの定義	12
3.2.2	陰関数 / 陰操作および宣言文の実装	13
3.2.3	メインプログラムの実装	14
3.2.4	生成された C++ コードの部分的置き換え	17
3.3	C++コードのコンパイル、リンク、実行	18
<b>4</b>	<b>サポートされていない構成要素</b>	<b>20</b>
<b>5</b>	<b>VDM 仕様のコード生成 - 詳細</b>	<b>22</b>
5.1	型のコード生成	22
5.1.1	動機付け	22
5.1.2	VDM-SL 型から C++へのマップ	25
5.1.3	VDM-SL 型名称のコード生成	30
5.1.4	不変条件	32
5.2	関数定義と操作定義のコード生成	33
5.3	値および状態定義のコード生成	34
5.4	値定義のコード生成	35
5.5	式と文のコード生成	36
5.6	名称仕様	37
5.7	標準ライブラリ	37
<b>A</b>	<b>libCG.a ライブラリ</b>	<b>39</b>
A.1	cg.h	39
A.2	cg_aux.h	40



<b>B</b>	<b>C++ 手書きファイル</b>	<b>41</b>
B.1	DefaultMod_userdef.h . . . . .	41
B.2	DefaultMod_userimpl.cc . . . . .	42
B.3	sort_ex.cc . . . . .	43
<b>C</b>	<b>Make ファイル</b>	<b>46</b>
C.1	Unix プラットフォームのための Make ファイル . . . . .	46
C.2	Windows プラットフォームのための Make ファイル . . . . .	49

# 1 導入

VDM-SL to C++ Code Generator は、VDM-SL 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-SL 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-SL 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 [3] に記載されている。本書は *VDMTools* ユーザマニュアル ( *VDM-SL* ) [6] の拡張であり、VDM-SL to C++ Code Generator への導入を行う。

コードジェネレータは、VDM-SL 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-SL 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-SL 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-SL 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-SL と C++ のデータ型間の関連を説明し、VDM-SL to C++ Code Generator 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-SL 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 [5] 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-SL 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-SL 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は、正確なコード生成のために、VDM-SL 仕様中の全モジュールの構文チェックがなされていることを要求する。

さらにコードジェネレータのみ、正しい型のモジュールに対して、コード生成を行うことができる。<sup>1</sup>コード生成を行う前に、そのモジュールが型チェックされていない場合は、Toolbox により自動的に型チェックが行われる。

### 2.2 グラフィカルインターフェイスの利用


ここでは VDM-SL 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。

VDM-SL 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

---

<sup>1</sup> [4] で説明されているように、よく形式化された2つのクラスが存在する。現内容では、可能な限りよく形式化された型正確性を意味する。

ファイル `sort.rtf` 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、[6] 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 the module *DefaultMod* 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。

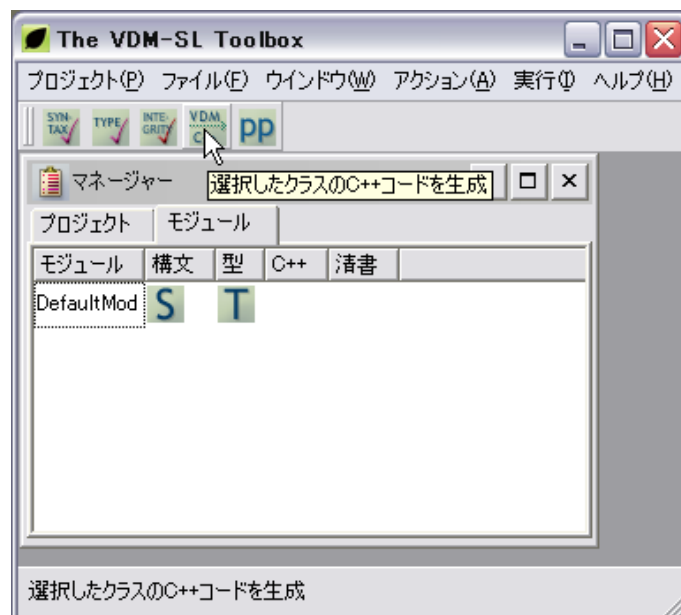


図 1: ソート例題のコード生成

仕様中の各 module - in this example only one - 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-SL 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。

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



図 2: ソート例題のコード生成

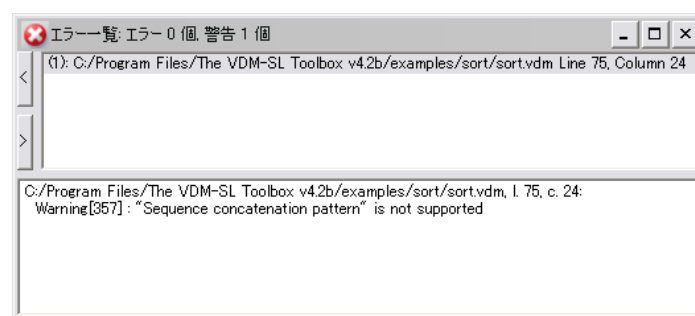


図 3: コードジェネレータで生成された警告

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-SL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。



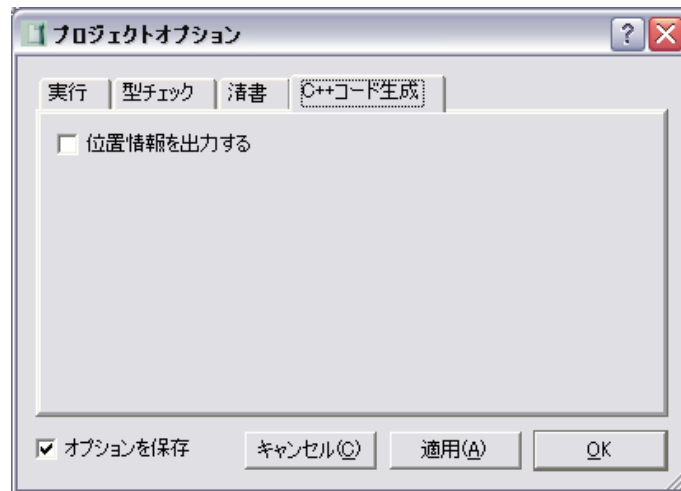


図 4: 実行時エラーで生成位置情報に対するオプション

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: sort.vdm. At line: 43 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-SL 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-SL Toolbox は、コマンドラインから vdmde 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:

```
vdmde -c [-r] 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 `vdmhome/examples/sort` ディレクトリで実行する:

```
vdmde -c sort.vdm
```

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。モジュールの各々に対して4つのファイルが生成される:

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

`<ModuleName>.h` ファイルは、仕様の VDM モジュールで定義されたすべての関数および操作に相当する C++ 関数宣言を含む。さらに、このモジュールで定義された合成型に相当するクラス定義を含む。

`<ModuleName>.cc` ファイルは、仕様の VDM モジュールで定義された関数および操作の実装を含む。さらに、レコード型に対し生成されるすべてのクラスに対する要素関数の実装がここにある。`<ModuleName>_anonym.h` および `<ModuleName>_anonym.cc` ファイルの目的は、仕様のモジュール中に現れるすべての匿名型を宣言し実装することにある。

VDM-SL to C++ Code Generator は、VDM-SL の構造化仕様およびフラット仕様 ([4] 参照) の両コード生成をサポートしている。ただし、フラット仕様は

DefaultMod という名称の既定モジュールに変換されている。このモジュールは、すべての VDM-SL 構成概念 (モジュール状態は除いて) をエクスポートする。

各 VDM-SL module 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 module 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-SL 仕様からたくさんの C++ ファイルが生成されるまで到達した。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らか基本的な知識が必要となる。これはまず最初に、VDM-SL 構成要素、特に VDM-SL 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-SL 型のコード生成 - 基本

この章では、VDM-SL 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 real -> bool
```

DefaultMod モジュールで定義されていて、以下のようにコード生成される:

```
class type_rL : public SEQ<Real>{
```

---

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

```
...
};

Bool vdm_DefaultMod_IsOrdered(const type_rL &vdm_DefaultMod_1){
...
};
```

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、いくら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++ Library 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。たとえば次のモジュール `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。

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-SL 型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 モジュール名が続き、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-SL 仕様と定義型の生成では、用いられた名称仕様を見よう：

```
module M
  exports all
  definitions
  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 モジュール `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 モジュール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: `<ModuleName>_anonym.{cc,h}`

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 モジュール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_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。

### 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-SL 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-SL モジュール単位の C++ ファイルと VDM-SL 仕様単位の C++ ファイルに分けることができる。

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 モジュールに対して、レコードタグに用いるオフセットを定義する。
2. VDM-SL 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。

---

<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。

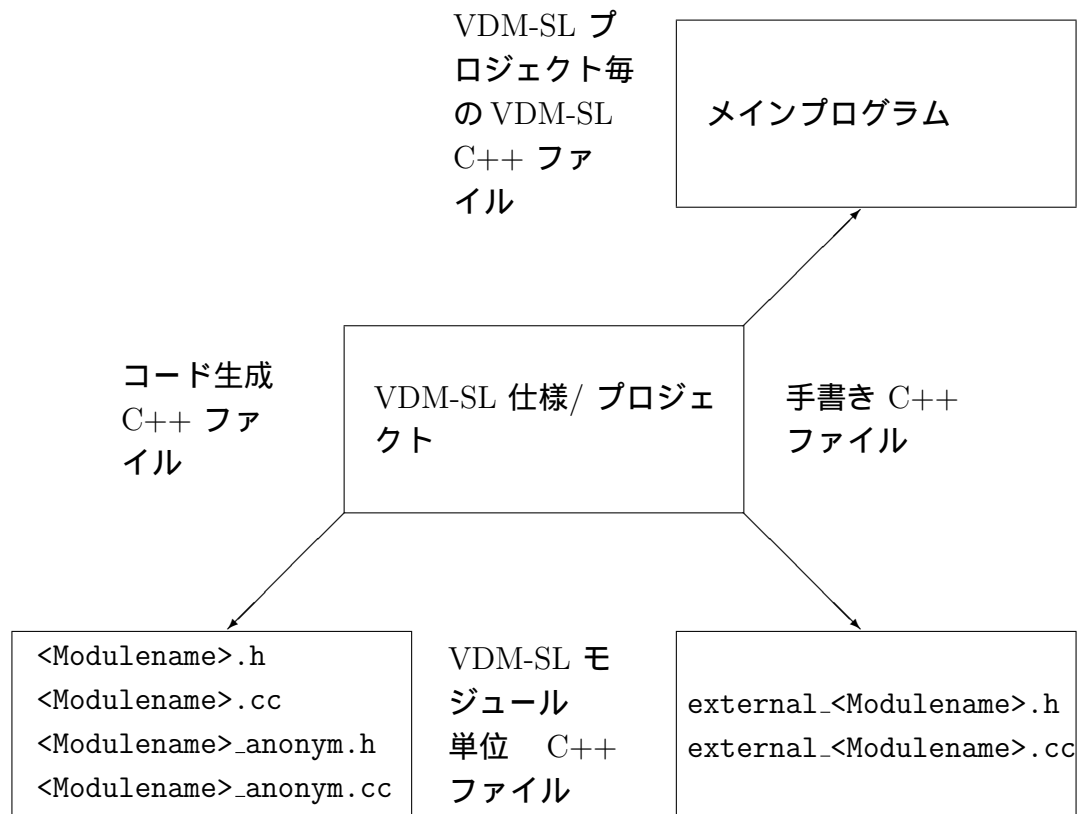


図 5: VDM-SL プロジェクト単位の C++ファイル

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-SL 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モジュールに対し、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、各モジュール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、`<ModuleName>_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_<ModuleName>.` 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、`DefaultMod_userdef.h` という名称の1つのファイルを実装する必要がある。このファイルには、以下のような定義が含まれるであろう。

```
#define TAG_DefaultMod 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モジュールに対して、その関数定義を含んだ `<ModuleName>_userimpl.cc` というファイルが書かれる必要がある。

このソート例題に、`ImplSort` という名称の1つの陰関数が含まれている。この関数は、生成されたC++コードとのインターフェイスとして、`DefaultMod_userimpl.cc` ファイルに実装されなければならない。関数は、生成されたC++関数宣言と一致するように、書かれなければならない。これは `DefaultMod.h` ファイル中に見つけ出すことができる：

```
type_rL vdm_DefaultMod_ImplSort(const type_rL &);
```

`DefaultMod` モジュールにある関数 `ImplSort` には、`vdm_DefaultMod_ImplSort` という名称が与えられる。

この関数の可能な実装例が1つ、付録 B.2 に載せてある。

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 モジュールを <ModuleName>\_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\_<ModuleName>\_<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。

最初に、VDM-SL メインプログラムの仕様を定めることから始めよう。

```
01    Main: () ==> ()
02    Main() ==
03    let l = [0, -12, 45] in
04    ( dcl res : seq of real,
05      b  : bool;
06      res := DoSort(l);
07      res := ExplSort(l);
08      res := ImplSort(l);
09      b  := post_ImplSort(l, res);
10      b  := inv_PosReal(hd l);
11      b  := inv_PosReal(hd res);
12      res := MergeSort(l))
```

ここで、上記仕様 VDM-SL メソッドにおけるのと同様の機能性をもつ C++ メインプログラムの実装を行うこととする。メインプログラムは sort\_ex.cc ファイル中に実装されて、付録 B.3 に全プログラムが載せられている。

メインプログラムを含める C++ ファイルは、必要な全ヘッダーファイルを含めることから始めるべきだろう。これらは VDM-SL モジュールごとに1つのヘッ

ダーファイルを含むが、VDM C++ Library のヘッダー、これは `metaiv.h` という名称、そして標準ライブラリクラスの `<fstream>` (これは出力を生成するため) である。

```
// Initialize values in DefaultMod
init_DefaultMod();
```

ここで、順に、上記の VDM 仕様を C++へ翻訳しよう。行 03 で実数の並びを指定している。C++に翻訳されると、以下のコードとなる:

```
type_rL l;
l.ImpAppend(Int(0));
l.ImpAppend(Int(-12));
l.ImpAppend(Int(45));
```

行 04 で、型 `seq of real` の変数 `res` を宣言するが、これは後でソートされた並びを含めるために用いられることになる。行 05 で型 `bool` の変数を宣言する。これに対する C++ コードは次である:

```
type_rL res;
Bool b;
```

ここで、仕様のソートメソッドをどのように呼び出すかを見よう。

行 06 は、引数として宣言された実数の並びとともに、`DoSort` 関数を呼び出す。結果はソートされた列となる。

コード生成された全 VDM-SL 型は、それぞれの VDM 値の ASCII 表現を含めた文字列を返す `ascii` メソッドをもつ。このメソッドはここでは、実行中に `cout`(標準出力) に関連するログメッセージを書き出すために用いられている。

```
cout << "Evaluating DoSort(" << l.ascii() << "):\n";
res = vdm_DefaultMod_DoSort(l);
cout << res.ascii() << "\n\n";
```

関数 `ExplSort` あるいは `ImplSort` とともに `l` をソートするため、同様に以下のコードの書き出しも可能である:

```
cout << "Evaluating ExplSort(" << l.ascii() << "):\n";
res = vdm_DefaultMod_ExplSort(l);
cout << res.ascii() << "\n\n";

cout << "Evaluating ImplSort(" << l.ascii() << "):\n";
res = vdm_DefaultMod_ImplSort(l);
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、関数 / 操作が陰であるか陽であるかに関係しないということである。

さらに、ImplSort に対してポスト条件関数を呼び出す場合を、想像してみることもできるであろう。コードジェネレータ はこれに対して `vdm_DefaultMod_post_ImplSort` という関数を生成してきた。この関数はまた通常の方法で呼び出すことができる。

```
cout << "Evaluating the post condition of ImplSort. \n"
      << "post_ImplSort(" << l.ascii() << ", "
      << res.ascii() << "):\n";
b = vdm_DefaultMod_post_ImplSort(l, res);
cout << b.ascii() << "\n\n";
```

VDM-SL 仕様や生成された C++ から分かるように、コードジェネレータ は型 `PosReal` に対して定義された不変数に対して不変数関数を生成してきた。この関数は `vdm_DefaultMod_inv_PosReal` と呼ばれ、通常の方法で呼び出すことができる:

```
cout << "Evaluation the invariant of PosReal. \n"
      << "inv_PosReal(" << l.Hd().ascii() << "):\n";
b = vdm_DefaultMod_inv_PosReal(l.Hd());
cout << b.ascii() << "\n\n";

cout << "Evaluation the invariant of PosReal. \n"
      << "inv_PosReal(" << res.Hd().ascii() << "):\n";
b = vdm_DefaultMod_inv_PosReal(res.Hd());
cout << b.ascii() << "\n\n";
```

最後に、MergeSort 関数 (行 12) を呼び出すことができ、これには実行時エラーが含まれることになるだろう。

```
cout << "Evaluating MergeSort(" << l.ascii() << "):\n";
res = vdm_DefaultMod_MergeSort(l);
cout << res.ascii() << "\n\n";
```

記述したメインプログラムは、`sort.ex.cc` という名称のファイルに実装され、付録 B.3 に載せてある。

### 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、MergeSort 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_DefaultMod_MergeSort` 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 `DefaultMod.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_DefaultMod_MergeSort(const type_rL
                                &vdm_DefaultMod_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DefaultMod\_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DefaultMod\_userdef.h ファイルに追加される必要がある:

```
#define DEF_DefaultMod_MergeSort
// the MergeSort function in module DefaultMod is hand coded
```

陰関数 / 陰操作定義を含まないモジュールに対して、<ModuleName>\_userimpl.cc はオプションである。ファイルが作成される場合は、<ModuleName>\_userimpl.cc ファイルが既に存在することを示すために、ユーザーは<ModuleName>\_userdef.h ファイルにもう1行追加する必要がある:

```
#define DEF_DefaultMod_USERIMPL
```

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-SL to C++ Code Generator 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-SL to C++ Code Generator 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-SL to C++ Code Generator 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++ Library。このライブラリは VDM-SL to C++ Code Generator 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 については、付録 C で示す。メインプログラム `sort_ex` をコンパイルするために、`makesort_ex` とタイプ入力しなければならない。

ここまで行くと、メインプログラム `sort_ex` を実行することができる。出力は以下に載せている。MergeSort の実行中に、実行時エラーが起きてしまうことに注意しよう。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ことが原因で引き起こされたものである。生成されたコードに含まれた位置情報が、基調となる仕様におけるエラー原因を導いている。

```
$ sort_ex
Evaluating DoSort([ 0,-12,45 ]):
[ -12,0,45 ]

Evaluating ExplSort([ 0,-12,45 ]):
[ -12,0,45 ]

Evaluating ImplSort([ 0,-12,45 ]):
[ -12,0,45 ]

Evaluating the post condition of ImplSort.
post_ImplSort([ 0,-12,45 ], [ -12,0,45 ]):
true

Evaluation the invariant of PosReal.
inv_PosReal(0):
true

Evaluation the invariant of PosReal.
```

```
inv_PosReal(-12):  
false  
  
Evaluating MergeSort([ 0, -12, 45 ]):  
Last recorded position:  
In: sort.vdm. At line: 43 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-SL 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}
```

- 文:
  - call 文中の ‘using’ 。
  - always 文、exit 文、 trap 文、 recursive trap 文。



- 次における型束縛 ([4] 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。

- 局所的関数定義

- より高順位の関数定義

- 関数値

- パラメータ化されたモジュール

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、モジュール、型、関数、操作、states、値、式、そして文、といった VDM-SL 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-SL 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 型のコード生成

第 3.1 章ですでに、VDM-SL 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 5.1.1 章では、VDM-SL 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 5.1.2 章で、各 VDM-SL 型から C++ コードへのマップを説明する。第 5.1.3 章では、型に対して用いる名称仕様をまとめている。

#### 5.1.1 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++ Library で見られる各 VDM-SL 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++ Library 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、`module A` における `char+` の `seq` は、`module 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。

```
module A
  exports all
  definitions
  types
    C = seq of char
end A
```

```
module B
  exports all
  definitions
  types
    D = seq of char
end B
```

VDM-SL 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-SL 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。

第一の問題は、`<ModuleName>anonym` ファイルを生成することで解決される。`<Module Name>_anonym.h` ファイルは、潜在的に他のモジュールでも(匿名型が)宣言されるような全型に対する型宣言を含む。`<ModuleName>_anonym.cc` はこれらの型の実装を含める。さらには、それらに対するマクロ定義を含める。匿名型でない型、つまり合成型や型名称は、`<ModuleName>_anonym.h` ファイルのかわりに `<Module Name>.h`

前に挙げた例題中で、`module 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。

### 5.1.2 VDM-SL 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
```

```
#ifndef 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-SL 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、module 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 GetTypeNames 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-SL 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-SL 値である (型ではない) ことだ。

### 5.1.3 VDM-SL 型名称のコード生成

VDM-SL 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-SL が構造等価を用いるため異なる。VDM-SL 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_ModuleName_A type_iL
#define TYPE_ModuleName_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、module 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-SL

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。

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  $\langle tp \rangle$  '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
<code>bool</code>	<code>b</code>	<code>b</code>
<code>nat1</code>	<code>i</code>	<code>i</code>
<code>nat</code>	<code>i</code>	<code>i</code>
<code>int</code>	<code>i</code>	<code>i</code>
<code>real</code>	<code>r</code>	<code>r</code>
<code>rat</code>	<code>r</code>	<code>r</code>
<code>char</code>	<code>c</code>	<code>c</code>
<code>quote</code>	<code>Q</code>	<code>&lt;Hello&gt;</code> は <code>Q</code> に変換される
<code>token</code>	<code>T</code>	<code>token</code> は <code>T</code> に変換される
<code>set</code>	$\langle tp \rangle S$	<code>set of char</code> は <code>cS</code> に変換される
<code>sequence</code>	$\langle tp \rangle L$	<code>sequence of real</code> は <code>rL</code> に変換される
<code>map</code>	$\langle tp1 \rangle \langle tp2 \rangle M$	<code>map set of int to char</code> は <code>iScM</code> に変換される
<code>product</code>	$\langle tp1 \rangle \dots \langle tpn \rangle \langle n \rangle P$	<code>int * char * sequence of real</code> は <code>icrS3P</code> に変換される
<code>composite</code>	$\langle length \rangle \langle name \rangle C$	合成型 <code>Comp</code> は <code>4CompC</code> に変換される。 $\langle length \rangle$ は合成型の名称に含まれる文字数であることに注意。
<code>union</code>	<code>U</code>	<code>int   char   real</code> は <code>U</code> に変換される
<code>optional</code>	$\langle tp \rangle 0$	<code>[ int ]</code> は <code>i0</code> に変換される

VDM	変換	使用例
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、 $A = \text{map int to } A$  における型 A は型名称 iFM で表現する。型定義  $B = \text{sequence of } A$  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  $\text{Comp} :: \dots$  は型 4CompC で表現される。

#### 5.1.4 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 ([2] 参照) と同じスコープ内で呼び出すことができる。VDM-SL to C++ Code Generator 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-SL 型定義を考えてみよう:

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

VDM-SL 関数 `inv_S` に相当するプロトタイプを、以下に示してある。`S` はモジュール `A` において定義されていると仮定する:

```
Bool vdm_A_inv_S(Set);
```

不変数関数の (プロトタイプの) 宣言は常にヘッダーファイル上に置かれ、したがって現在は全型がエクスポートされる。

VDM-SL to C++ Code Generator 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。

## 5.2 関数定義と操作定義のコード生成

VDM-SL to C++ Code Generator は、陰と陽両方の関数 / 操作の C++ プロトタイプを生成する。ある関数<sup>4</sup> をそれを定義するモジュールからエクスポートする場合は、相当する C++ プロトタイプがヘッダーファイルに置かれることになる。エクスポートされたのではない関数は、実装ファイル中に `static` 関数として宣言されている。

### 陽関数 / 陽操作の定義

VDM-SL to C++ Code Generator は、すべての陽の VDM-SL 関数 / 操作に対する C++ 関数定義を生成する。これらの関数定義は実装ファイル中に置かれる。

### 陰関数 / 陰操作の定義

コードジェネレータ は、陰関数に相当する C++ 関数プロトタイプに加えて、実装ファイル中にインクルードプリプロセッサを生成する。もしモジュール `A` が e.g. たとえばいくつかの陰関数を含むとすると、以下のプリプロセッサがモジュール `A` (`A.cc`) の実装ファイル中に現れるはずである:

---

<sup>4</sup>以下では、関数とは関数と操作の両方を参照するものとする。

```
#include "A_userimpl.cc"
```

したがって、陰関数を `A_userimpl.cc` ファイル中に実装することは、ユーザーの責任に任される。このファイルが生成されていない場合、コンパイル時エラーとなることに注意しよう。

### 事前条件 / 事後条件

事前条件 / 事後条件が指定されると、それに相当する事前関数 / 事後関数が暗黙に定義される ([4] 参照)。これらの関数それぞれに対して、C++ プロトタイプと C++ 関数が生成される。事前関数 / 事後関数は暗黙にエクスポートされ、そのためそれらに相当する VDM-SL 関数定義がエクスポートされる場合はヘッダーファイルに置かれる。

VDM-SL to C++ Code Generator の最新版は事前 / 事後条件の動的チェックをサポートしないことには注意しよう。これらのチェックをうけるために、事前 / 事後関数が明白に呼び出されなければならない。

## 5.3 値および状態定義のコード生成

状態および値定義に相当するコード生成は、直接的である。各値や各状態構成要素に対して、C++変数が宣言される。エクスポートされないすべての状態構成要素や値は、実装ファイル中 `static` 変数として宣言されている。各エクスポートされた値に対して、ヘッダーファイルに `extern` 変数宣言が置かれる。

加えて、状態定義はレコード型定義として取り扱われる (第 5.1.2 章参照)。

### 初期化関数

値と状態構成要素の初期化は、`init_module` という名称の関数によってなされる。初期化述語をコード生成するためには、インタプリタによって要求される形式でなければならない ([4] 参照)。この例外の場合は、コードジェネレータが“インクルード文”を生成するが、初期化のユーザー定義された定義を含める。以下の、モジュール A の状態定義を考えよう：

```
state Sigma of
  a : int
  b : bool
init s == s = mk_Sigma(10, false)
end
```

この場合は、初期化述語がインタープリタに要求される形式であり、状態の初期化は完全に行われることになる。もし初期化述語がこの形式でないとすると、コードジェネレータが `init_A` 内に以下のインクルード命令を生成することになるであろう

```
{
#include "A_init.cc"
}
```

このファイルの存在を確保することは、ユーザーの責任である。

## 5.4 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。

状態と値の定義に相当するコードは、直接的である。各値や各状態構成要素に対して、C++変数が宣言される。エクスポートされていないすべての状態構成要素と値は、実装ファイル中の `static` 変数として宣言される。各エクスポートされた値に対しては、`extern` 変数宣言がヘッダーファイル中に置かれる。

モジュール A の仕様を考えてみよう:

```
module A

exports
  values b: int

definitions
```

```
values
  mk_(a,b) = mk_(3,6);
  c :char = 'c'
end A
```

相当する *A.cc* ファイルを以下に並べる:

```
#include "A.h"
static Int vdm_A_a;
Int vdm_A_b;
static Char vdm_A_c;

....

void init_A() {
  ...
  //Code that initialises the values in the specification.
}
```

*vdm\_A\_a* および *vdm\_A\_c* は静的変数であるが、一方エクスポートされた値 *vdm\_A\_b* はそうでない。

A モジュール (*A.h*) の相当するヘッダーファイルは、以下に述べた通り:

```
extern Int vdm_A_b;
void init_A();
```

*vdm\_A\_b* は *extern* 変数として宣言されることに注意しよう。

関数 *init\_A()* は、ユーザーによりメインプログラム中で呼び出されるべきである。

## 5.5 式と文のコード生成

VDM-SL 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。

## 5.6 名称仕様

仕様中の変数は、C++ 生成コード中の変数に翻訳される。VDM-SL to C++ Code Generator で用いられる名称付け法は、これら全変数の名称替え: `vdm_module_name`

とすることで、ここで `module` は `name` が定義されたモジュールの名称である。関数 `f` e.g. がモジュール `M` で定義された場合、`f` に相当する C++ 関数は `vdm_M_f` と名称付けされる。加えて以下の名称がコードジェネレータで使われる:

- `init_module`: モジュール `module` の値と状態を初期化する関数。
- `length_module_record`: レコード `record` の項目数を定義している `static` 変数で、モジュール `module` 内で定義されている。
- `pos_module_record_field`: レコード `record` 中の項目選択枝 `field` の位置 / 添え字 (整数) を定義している 静的な 変数で、モジュール `module` 内で定義されている。
- `name_number`: 生成 C++ コードで用いられた一時的な変数。

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。

## 5.7 標準ライブラリ

### Math ライブラリ

Math ライブラリ (`math.vdm` respectively `mathflat.vdm` 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 `vdmhome/cg/include` にある。

## IO ライブラリ

IO ライブラリ (the `io.vdm` respectively `ioflat.vdm` 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 `vdmhome/cg/include` にある。

IO ライブラリの `freadval` 関数からなる利用では、`module` 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 5.4 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、`module 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. *The VDM++ Language*. CSK.
- [3] CSK. *VDM-SL Installation Guide*. CSK.
- [4] CSK. *The VDM-SL Language*. CSK.
- [5] CSK. *VDM-SL Sorting Algorithms*. CSK.
- [6] CSK. *VDM-SL 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-SL 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-SL データ型<sup>5</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。

vdmBase クラスは、コードジェネレータの VDM<sup>++</sup> 版でコード生成される場合にのみ用いられる。

---

<sup>5</sup>この版の VDM-SL to C++ Code Generator では VDM C++ Library の使用のみが可能である。

```
/**
 * * 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

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 DefaultMod\_userdef.h

```
#define TAG_DefaultMod 100
```

## B.2 DefaultMod\_userimpl.cc

MergeSort の手書き版としては、vdm\_DefaultMod\_ImplSort の実装を選択する。

```
/**
 * * WHAT
 * *   Handwritten implementation of 'merge sort'
 * * ID
 * *   $Id: DefaultMod_userimpl.cc,v 1.4 2005/05/13 00:41:46 vdmtools Exp $
 * * PROJECT
 * *   Toolbox
 * * COPYRIGHT
 * *   (C) 2016 Kyushu University
 */

static type_rL Merge(type_rL, type_rL);

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

type_rL Merge(type_rL l1, type_rL l2)
{
    if (l1.Length() == 0)
```

```
    return l2;
else if (l2.Length() == 0)
    return l1;
else {
    type_rL 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.3 sort\_ex.cc

```
/**
 * * WHAT
 * *   Main C++ program for the VDM-SL sort example
 * * ID
 * *   $Id: sort_ex.cc,v 1.9 2005/05/27 07:47:27 vdmtools Exp $
 * *   This file is distributed as sort_ex.cpp under win32.
 * * PROJECT
 * *   Toolbox
 * * COPYRIGHT
 * *   (C) 2016 Kyushu University
 */

#include <fstream>
#include "metaiv.h"
#include "DefaultMod.h"

int main(int argc, const char * argv[])
{
    // Initialize values in DefaultMod
```

```
init_DefaultMod();

// Constructing the value l = [0, -12, 45]
type_rL l (mk_sequence(Int(0), Int(-12), Int(45)));

type_rL res;
Bool b;

// res := DoSort(l);
wcout << L"Evaluating DoSort(" << l.ascii() << L"):" << endl;
res = vdm_DefaultMod_DoSort(l);
wcout << res.ascii() << endl << endl;

// res := ExplSort(l);
wcout << L"Evaluating ExplSort(" << l.ascii() << L"):" << endl;
res = vdm_DefaultMod_ExplSort(l);
wcout << res.ascii() << endl << endl;

// res := ImplSort(l);
wcout << L"Evaluating ImplSort(" << l.ascii() << L"):" << endl;
res = vdm_DefaultMod_ImplSort(l);
wcout << res.ascii() << endl << endl;

// b := post_ImplSort(l, res);
wcout << L"Evaluating the post condition of ImplSort." << endl;
wcout << L"post_ImplSort(" << l.ascii() << L", " << res.ascii() << L"):" << endl;
b = vdm_DefaultMod_post_ImplSort(l, res);
wcout << b.ascii() << endl << endl;

// b := inv_PosReal(hd l);
wcout << L"Evaluation the invariant of PosReal." << endl;
wcout << L"inv_PosReal(" << l.Hd().ascii() << L"):" << endl;
b = vdm_DefaultMod_inv_PosReal(l.Hd());
wcout << b.ascii() << endl << endl;

// inv_PosReal(hd res);
```



```
wcout << L"Evaluation the invariant of PosReal." << endl;
wcout << L"inv_PosReal(" << res.Hd().ascii() << L"):" << endl;
b = vdm_DefaultMod_inv_PosReal(res.Hd());
wcout << b.ascii() << endl << endl;

// res := MergeSort(l);
// This will imply a run-time error!
wcout << L"Evaluating MergeSort(" << l.ascii() << L"):" << endl;
res = vdm_DefaultMod_MergeSort(l);
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-SL 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 macro VDMHOME to fit to your directory
# structure.
#
# Note that this version of the code generator must be used
# with egcs version 1.1
#

OSTYPE=$(shell uname)

VDMHOME = ../..
VDMDE   = $(VDMHOME)/bin/vdmde

INCL     = -I $(VDMHOME)/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      = $(CCPATH)clang++
else
ifeq ($(strip $(OSMV)),11) # 10.7
```

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

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

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

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

```
endif

CCFLAGS = -g -Wall

OSTYPE2=$(word 1, $(subst _, ,$(strip $(OSTYPE))))
ifeq ($(strip $(OSTYPE2)),CYGWIN)
all: winmake
else
all: sort_ex
endif

winmake:
make -f Makefile.winnt

sort_ex: sort.o sort_ex.o
${CC} ${CCFLAGS} -o sort_ex sort_ex.o sort.o ${LIB}

sort_ex.o: sort_ex.cc
${CC} ${CCFLAGS} -c -o sort_ex.o sort_ex.cc ${INCL}

#external_DefaultMod.cc
sort.o: DefaultMod.h DefaultMod.cc
${CC} ${CCFLAGS} -c -o sort.o DefaultMod.cc ${INCL}

DefaultMod.h DefaultMod.cc \
DefaultMod_anonym.h DefaultMod_anonym.cc: sort.vdm
$(VDMDE) -c $^

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

SPECFILE = sort.vdm

VDMLoop = vdmloop
```

```
GENFILES = sort.vdm.tex sort.vdm.log sort.vdm.aux sort.vdm.dvi \  
           sort.vdm.ps vdm.tc DefaultMod.h DefaultMod.cc \  
           DefaultMod_anonym.h DefaultMod_anonym.cc
```

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

```
%.tex: $(SPECFILE) vdm.tc  
vdmde -lrNn $(SPECFILE)
```

```
sort.vdm.ps: $(SPECFILE).tex  
latex sort.vdm.tex  
latex sort.vdm.tex  
dvips sort.vdm.dvi -o
```

```
#####  
#### Clean                                     ####  
#####
```

```
clean:  
rm -f *~ *.o sort.o sort_ex.o sort_ex m4tag_rep  
rm -f $(GENFILES)  
rm -f *.cpp *.obj *.exe  
cd test; rm -f vdm.tc *.res
```

## C.2 Windows プラットフォームのための Make ファイル

```
#  
# WHAT  
#   Windows CYGWIN GNU makefile for the code generated  
#   VDM-SL sort example.  
# ID  
#   $Id: Makefile.winnt,v 1.14 2005/12/21 08:41:00 vdmtools Exp $  
# PROJECT
```



```
#    Toolbox
# COPYRIGHT
#    (C) 2016 Kyushu University
#
#

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

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

VDMDE = "$ (TBDIR)/bin/vdmde.exe"

CC = cl.exe
LINK = link.exe

CFLAGS = /nologo /c /MD /W0 /GD /EHsc /TP

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

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

## CG Version Files

CGSOURCES = DefaultMod.cpp DefaultMod.h DefaultMod_anonym.cpp DefaultMod_anonym.h

CGOBSJS = DefaultMod.obj

CGFLAGS = /D CG

sort_ex.exe: sort_ex.obj $ (CGOBSJS)

sort_ex.obj: sort_ex.cpp DefaultMod.h

DefaultMod.obj: DefaultMod.cpp DefaultMod.h DefaultMod_anonym.cpp \
```

```
DefaultMod_anonym.h DefaultMod_userdef.h \  
DefaultMod_userimpl.cpp
```

```
SPECFILES = sort.rtf
```

```
$(CGSOURCES): $(SPECFILES)  
$(VDMDE) -c $^
```

```
all: sort_ex.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 = sort.rtf.rtf vdm.tc
```

```
vdm.tc:  
cd test  
$(VDMLOOP)  
cd ..
```



```
cp test\${@} .\
```

```
sort.rtf.rtf: $(SPECFILES) vdm.tc  
"${VDMDE}" -lrNn $(SPECFILES)
```

```
clean:
```

```
rm -f *.obj DefaultMod.obj sort_ex.obj sort_ex.exe m4tag_rep  
rm -f $(GENFILES) DefaultMod.h DefaultMod.cpp sort_ex.cpp  
rm -f DefaultMod_userimpl.cpp  
rm -f DefaultMod_anonym.cpp DefaultMod_anonym.h
```