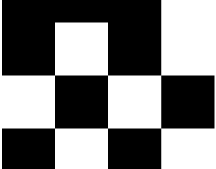

VDMTools®

The VDM++ to C++ Code Generator

IFAD

How to contact IFAD:

☎	+45 63 15 71 31	Phone
	+45 65 93 29 99	Fax
✉	IFAD Forskerparken 10A DK - 5230 Odense M	Mail
	http://www.ifad.dk ftp.ifad.dk	Web Anonymous FTP server
@	toolbox@ifad.dk info@ifad.dk sales@ifad.dk	Technical support General information Sales and pricing

The VDM++ to C++ Code Generator — Revised for V6.8

© COPYRIGHT 2001 by IFAD

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

This document is subject to change without notice

Contents

1 Introduction 1

2 Invoking the Code Generator 1
2.1 Requirements for Generating Code 2
2.2 Using the Graphical Interface 2
2.3 Using the Command Line Interface 5
2.4 Generated C++ Files 6

3 Interfacing the Generated Code 7
3.1 Code Generating VDM++ Types - The Basics 7
3.2 Files to be Implemented by the User 11
3.2.1 Definition of Offsets for Record Tags 12
3.2.2 Implementing Implicit Functions/Operations and Specification Statements 13
3.2.3 Implementing the Main Program 14
3.2.4 Substituting Parts of the Generated C++ code 17
3.3 Compiling, Linking and Running the C++ code 18

4 Unsupported Constructs 19

5 Code Generating VDM Specifications - The Details 21
5.1 Code Generating Classes 21
5.1.1 Object References in the VDM C++ Library 22
5.1.2 The Inheritance Structure of the Generated Code of Classes 23
5.1.3 The Structure of a Generated Class 27
5.2 Code Generating Types 28
5.2.1 Motivation 28
5.2.2 Mapping VDM++ Types to C++ 30
5.2.3 Code Generating VDM++ Type Names 36
5.2.4 Invariants 38
5.3 Code Generating Function and Operation Definitions 39
5.4 Code Generating Instance Variables 42
5.5 Code Generating Value Definitions 44
5.6 Code Generating Expressions and Statements 45
5.7 Name Conventions 45
5.8 Standard Library 46

A References 47

B The libCG.a Library **47**
B.1 cg.h 47
B.2 cg_aux.h 49

C Handcoded C++ Files **50**
C.1 DoSort_userdef.h 50
C.2 ExplSort_userdef.h 50
C.3 ImplSort_userdef.h 50
C.4 MergeSort_userdef.h 50
C.5 SortMachine_userdef.h 50
C.6 Sorter_userdef.h 50
C.7 ImplSort_userimpl.cc 51
C.8 sort_pp.cc 52

D Makefiles **55**
D.1 Makefile for Unix Platform 55
D.2 Makefile for Windows Platform 57

1 Introduction

The VDM++ to C++ Code Generator supports automatic generation of C++ code from VDM++ specifications. This way, the Code Generator provides you with a fast way of implementing applications based on VDM++ specifications.

The Code Generator is an add-on feature to the VDM++ Toolbox. Its installation is described in the document [InstallPPMan]. The following text is an extension to the *User Manual for the IFAD VDM++ Toolbox* [UserManPP] and it gives you an introduction to the VDM++ to C++ Code Generator.

The Code Generator supports approximately 95% of all the VDM++ constructs. As a supplement, the user is given the possibility of substituting parts of the generated code with handwritten code.

This manual is structured in the following way:

Section 2 lists the requirements that the VDM++ specification has to satisfy in order to generate correct C++ code. Moreover, this section describes how to invoke the VDM++ to C++ Code Generator from the IFAD VDM++ Toolbox. Finally, the code generated C++ files will be described.

Section 3 guides you in the writing of an interface to the generated C++ code and it explains how to interface handwritten code to it. Furthermore, it will be explained how to compile, link and run the C++ code.

Section 4 summarizes the VDM++ constructs not supported by the Code Generator.

Section 5 gives a detailed description of the structure of the generated C++ code. In addition, it explains the relation between VDM++ and C++ data types, and it describes some of the design decisions made, when developing the VDM++ to C++ Code Generator, including the name conventions used. This section should be studied intensively before using the Code Generator professionally.

2 Invoking the Code Generator

To get started using the Code Generator you should write a VDM++ specification in one or several files. In the distribution of the Toolbox a specification of different sorting algorithms is included. This specification will be used in the following in order to describe the use of the Code Generator. This specification is described

in [SortEx]. It is recommended that you go through the described steps on your own computer. In order to do so, copy the directory `vdmhome/examples/sort` and `cd` to it.

Before generating C++ code, it has to be ensured, that the VDM++ specification satisfies the necessary requirements. The requirements in question will be described in Section 2.1. In Section 2.2 and 2.3 it will be explained how to generate C++ code using the VDM++ Toolbox from the graphical interface and from the command line. In Section 2.4 the code generated C++ files will be described.

2.1 Requirements for Generating Code

The Code Generator requires that all files of the VDM++ specification are syntax checked in order to generate correct code. That is, one can code generate a single class, however, all files of the specification should be checked.

Moreover, the Code Generator can only generate code for classes which are type correct.[1] If a class has not been type checked before and one tries to generate code for it, it is automatically type checked by the Toolbox.

2.2 Using the Graphical Interface

We will now describe how the sort example is code generated from the graphical user interface of the VDM++ Toolbox.

The VDM++ Toolbox is started with the command `vdmgde`. In order to generate code corresponding to the sort example, create a new project containing all the `*.rtf` files which can be found in the directory `/vdmhome/examples/sort`. See [UserManPP] for a description of how to configure a project.

The file(s) must first be syntax checked and type checked: if you don't do this manually, the Toolbox will do it automatically when the Code Generator is invoked. The sort example specification passes both checks with no errors. Then the Code Generator can be invoked by selecting all the classes and pressing the (Generate C++) button. More than one file/class can be selected, in which case all of them are translated to C++. The result of this step is shown in Figure 1.

Code is generated for each class in the specification and the Toolbox signals this

[1] There exist two classes of well-formedness as explained in [LangManPP]. In the current context we mean possible well-formed type correctness.

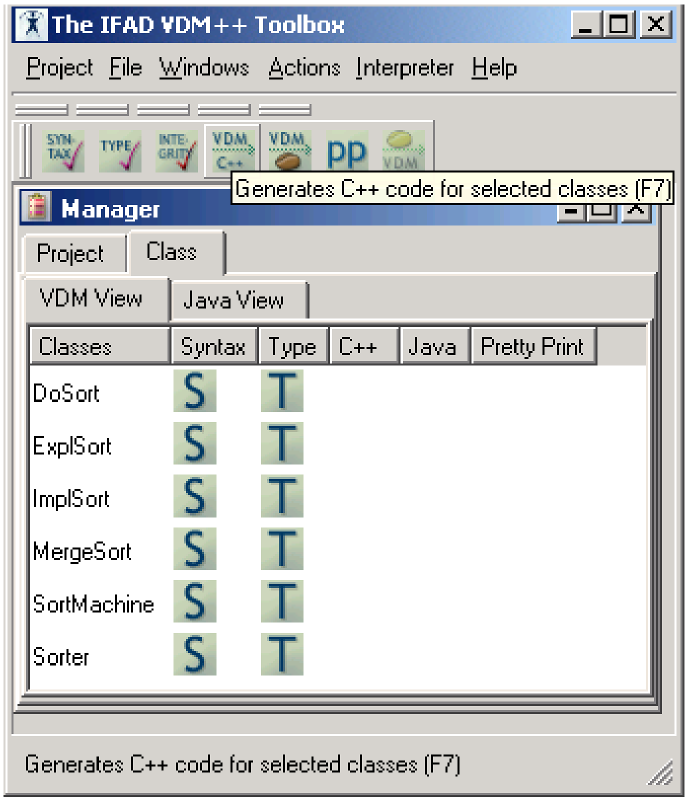

Figure 1: Code Generating the Sort Example

by writing a big *C* as shown in Figure 2. A number of C++ files have been created in the directory, where your project file lies. If no project file exists, the files will be written in the directory, where the VDM++ Toolbox was started.

When generating code for the sort example, one warning is generated by the Code Generator, and the *Error* window therefore pops up as shown in Figure 3.

This warning states that the sequence concatenation pattern is not supported by the Code Generator. This means, that the Code Generator cannot generate executable C++ code for this construct. The generated code will be compilable, but executing the branch containing the unsupported construct will cause a run-time error. A detailed list of unsupported constructs is given in Section 4.

The user of the Code Generator can choose to generate code containing position information of run-time errors. When the *Output position information* option is chosen, run-time error messages will tell you the position (file name, line and column number) in the VDM++ specification which causes the run-time error.

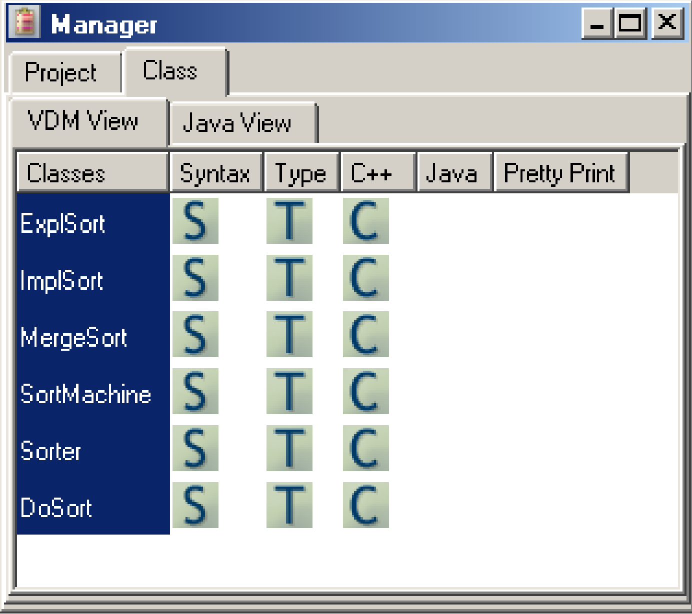

Figure 2: Code Generating the Sort Example

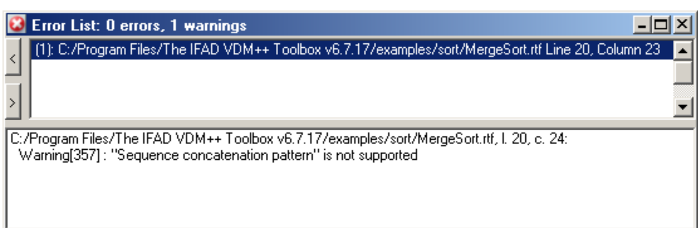

Figure 3: A Warning Generated by the Code Generator

This feature can be set in the option menu, as shown in Figure 4. Another option available is the *Check pre and post conditions* option. This generates code which checks operation and function pre conditions, and function post conditions. It is also shown in Figure 4.

For the sort example, the execution of the `MergeSort` function will, as described above, result in a run-time error. Without setting the described option, the execution of the corresponding C++ code will result in the following error message:

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

On the other hand, when setting the option, the following error message will

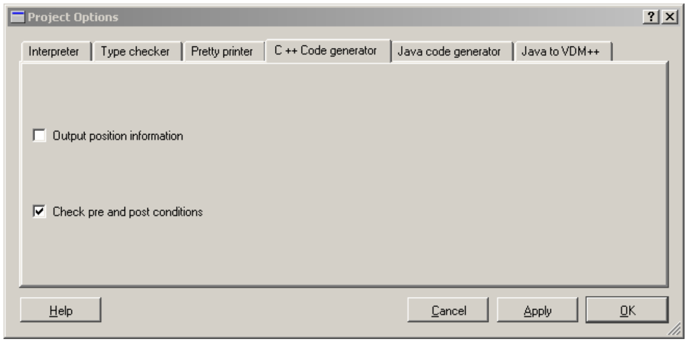

Figure 4: C++ Code Generator Options

appear:

```
Last recorded position:
In: MergeSort. At line: 31 column: 18
The construct is not supported: Sequence concatenation pattern
```

2.3 Using the Command Line Interface

The Code Generator can of course also be invoked when the VDM++ Toolbox is run from the command line. This will be described briefly in the following.

The VDM++ Toolbox is started from the command line with the command `vppde`. The -c option is used in order to generate code:

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

In order to code generate the sort example, the following command is executed in the `vpphome/examples/sort` directory:

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

The specification will be parsed first. If no syntax errors are detected, the specification will be type checked for possible well-formedness. Finally, if no type errors are detected, the specification will be translated into a number of C++ files. Corresponding to the graphical interface, the user can set the *Output position*

information option (`-r`) in order to generate code with run-time position information, and the *Check pre and post conditions* option (`-P`) to generate run-time checks of pre and post conditions.

2.4 Generated C++ Files

Let us now go one step further and look at the files generated by the code generator.

For each VDM++ class, four files are generated:

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

The `<ClassName>.h` file contains the definition of the C++ class corresponding to the VDM++ class. Moreover it contains the class definitions corresponding to the composite types defined in the VDM++ class.

The `<ClassName>.cc` file contains the implementation of the functions and operations defined in the VDM++ class. Moreover, for every class generated for a record type, the implementation of its member functions is found here.

The purpose of the `<ClassName>_anonym.h` and `<ClassName>_anonym.cc` files is to declare and implement all the anonymous types. Anonymous types are those that are not given a name in the VDM++ specification.

Apart from these files, two more files are generated:

- `CGBase.h`
- `CGBase.cc`

These files contain part of the implementation of the object reference type. This is described in Section 5.1.

The code corresponding to each VDM++ class is divided into a header and an implementation file. Both files will be named with the class name. The suffix

for the header file will be '`.h`', whereas the suffix for the implementation file will '`.cc`' on Unix platforms and '`.cpp`' on Windows platforms. The suffix of the implementation file can be customised by setting the environment variable **VDMCGEXT**[2].

3 Interfacing the Generated Code

We have now reached the point, where a number of C++ files have been generated from a VDM++ specification. You are now in the position to write an interface to these C++ files in order to compile, link and run an application.

To be able to write an interface to the generated code, you must have some basic knowledge about the generated code. This includes, first of all, the strategy used when generating code for VDM++ constructs, especially VDM++ types. In the following, we will give a short introduction to this topic. For more information the reader is referred to Section 5.

3.1 Code Generating VDM++ Types - The Basics

This section gives a short introduction to the way VDM++ types are code generated.

Let us start by giving an example of generated C++ code. The signature of the function `IsOrdered`,

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

defined in class `ExplSort`, is for example code generated as follows:

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

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

[2]On the Windows 98 platform this can be set in the `autoexec.bat` file, and on the Windows NT/2000 platform this can be set in the `Registry`

In order to understand this code, it is necessary to have some knowledge about the strategy used to generate code for VDM types, as well as the used name conventions.

The data type handling of the Code Generator is based upon the VDM C++ Library. The current version of this library (`libvdm.a`) is described in [LibMan].

- *Basic Data Types*

  The basic data types are mapped to the corresponding VDM C++ library classes, `Bool`, `Int`, `Real`, `Char` and `Token`.

- *Quote Types*

  The quote types are mapped into the corresponding VDM C++ library class `Quote`.

- *Set, Sequence and Map Types*

  To handle the compound types `set`, `sequence` and `map`, templates are introduced. These templates are also defined in the VDM C++ library. As an example let us show how the VDM type `seq of int` is code generated:

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

  The VDM `seq` type is mapped into a class that inherits from the template `SEQ` class. In case of `seq of int`, the argument of the template class is `Int`, the C++ class representing the basic VDM type `int`. The name of the new class is made up in the following way:

  `type` : signals an anonymous type
  `i`: signals int
  `L`: signals sequence

- *Composite/Record Types*

  Each composite type is mapped into a class that is a subclass of the VDM C++ library `Record` class. For example, the following composite type defined in class `M`

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

will be code generated as:

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

- *Tuple Types*

  The strategy for handling tuples is very similar to that of composite types. Each tuple type is mapped into a class that is a subclass of the VDM C++ library `Tuple` class. For example, the following tuple:

  ```
  int * real
  ```

  will be code generated as:

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

  The name of the new class is made up in the following way:

  `type` : signals an anonymous type
  `i`: signals int
  `r`: signals real
  2P: signals tuple with size two.

- *Union Types*

  The union type is mapped into the VDM C++ library `Generic` class.

- *Optional Types*

  The optional type is mapped into the VDM C++ library `Generic` class.

- *Object Reference Types*

  For each VDM++ class a corresponding C++ class is generated. For a VDM++ class *SortMachine* a corresponding C++ class *vdm_SortMachine* will be generated.

  An object reference of an instance of class is mapped into a class `type_ref_<ClassName>`.

  The object reference type is described in detail in Section 5.1.

The composite type example has already given you an idea about the way, type names are generated.

Types that are not given a name in the specification (anonymous types) are prefixed with `type` written with small letters. Type names however are prefixed with `TYPE` written with capital letters. In the record example we have seen one example of a type name, namely `TYPE_M_A`. The other VDM++ types can of course also be named and the name scheme used is the same as for records.

A generated type name is prefixed with `TYPE`. Then it is followed by the class name, where the type is defined, and finally the chosen VDM name is concatenated.

Take a look at the following VDM++ specification and the name conventions used when generating the defined types:

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

The three defined types above will be given the names `TYPE_M_A`, `TYPE_M_B` and `TYPE_M_C`. The scope of these type names is limited to the class `M`. The definition of these names is therefore placed in file *M.h*.

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

However, the specification also contains two anonymous types `int * real` and `seq of int`. These types can potentially be used in any class, and therefore the name and definition of the corresponding C++ type should be declared and defined globally[3]. This is done in the *anonym* files: `<ClassName>_anonym.{h, cc}`

[3]The strategy is to define unique names for structurally equal types. This strategy is used in order to solve the fact that C++ is based on type name equivalence, whereas VDM++ is based on structural equivalence.

In addition to the information given about the way types are code generated, it should be mentioned how function or operation names are generated: A function or operation name `f` in a class M in a VDM specification will be given the name: `vdm_M::vdm_f`.

You should now have an idea about the Code Generator's overall strategy when generating code for VDM specifications. More detailed information is given in Section 5 and should be studied carefully before using the Code Generator professionally.

3.2 Files to be Implemented by the User

We have now given some basic information about the Code Generator and the code generated files. This section describes the work the user must do in order to interface with the generated code.

Let us start by giving you an overview of all the C++ files involved when running the C++ code for a VDM++ specification. These files can be split up into code generated C++ files and handcoded C++ files. Figure 5 shows the code generated files to the left and the handcoded files to the right. Moreover, the involved files can be split up into C++ files per VDM++ class and C++ files per VDM++ specification.

Section 2.4 has already described the code generated C++ files. We will now decribe the handcoded files.

To interface with the generated code the user has to perform the following tasks:

1. For each class define offsets for record tags.
2. Implement implicit functions/operations and specification statements contained in the VDM++ specification.
3. Write a main program.
4. Optionally, substitute parts of the generated C++ code with handwritten code.
5. Compile, link and run the application.

In the following we will describe these tasks one by one for the sort example.

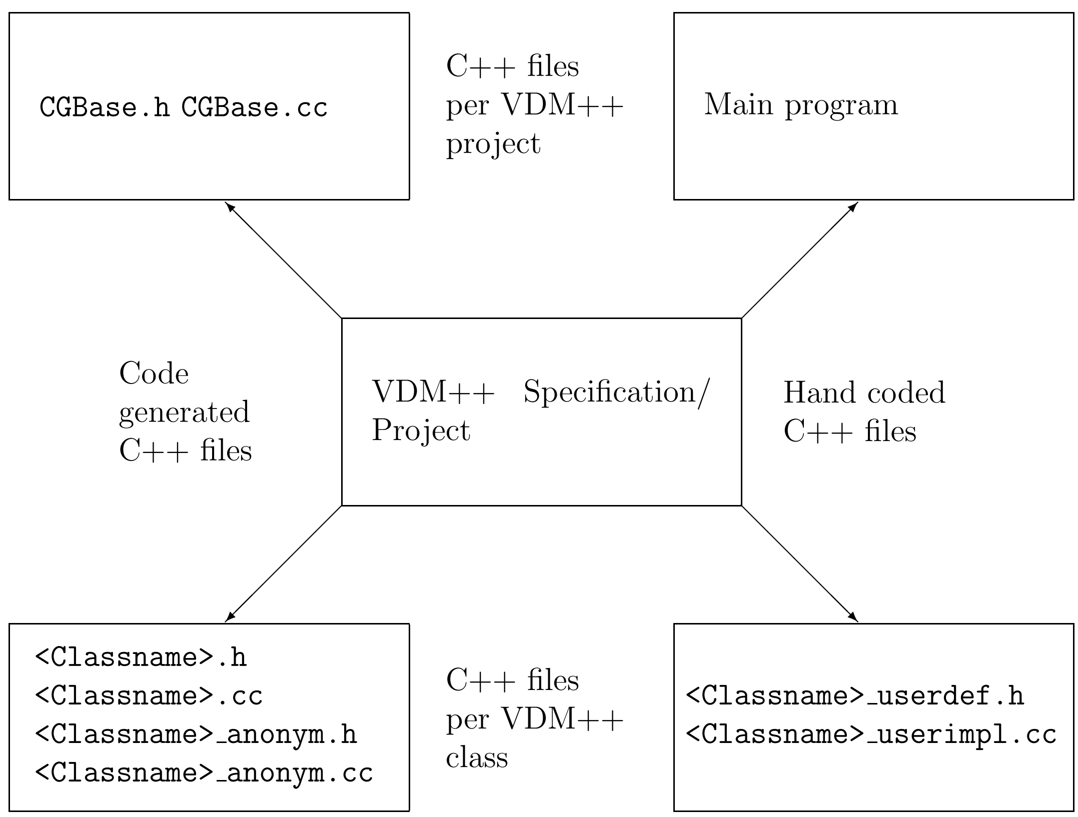

Figure 5: C++ Files per VDM++ Project

3.2.1 Definition of Offsets for Record Tags

A composite type (record type) consists in the VDM++ specification of a string (the tag) and a sequence of field selections for each field in the record type:

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

In the VDM C++ Library the record tag string *"RecTag"* is modelled with a unique integer. However, it is important for this strategy that all record types within one specification have their own unique tag number. For each class the Code Generator will number the record tags sequentially from an offset basis. The offset should be defined by the user, and it is the responsibility of the user that the offsets defined for each class ensures that the tags are unique.

The definition of the offset should be written in a file called `<ClassName>_userdef.h` The offset should be defined with a define directive. The name of the tag offset should be `TAG_<ClassName>`.

For the sort example this implies, that the user has to implement 6 files, one for each class. The `MergeSort_userdef.h` file, for example, could possibly contain the following definition.

```
#define TAG_MergeSort 100
```

3.2.2 Implementing Implicit Functions/Operations and Specification Statements

For every class which contains an implicit function a `<ClassName>_userimpl.cc` file containing the function definition has to be written.

The sort example contains one implicit function in the `ImplSort` class, namely `ImplSorter`. This function must be implemented in the file `ImplSort_userimpl.cc` in order to interface the generated C++ code. The function has to be written in such a way that it matches its member function declaration which can be found in class `vdm_ImplSort`, defined in file `ImplSort.h`:

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

The function `ImplSorter` in class `ImplSort` is given the name `vdm_ImplSort::vdm_ImplSorter` and has the following declaration:

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

One possible implementation of this function is shown in Appendix C.7.

Thus, the user has to write a C++ function definition for the operation and add it to the `<ClassName>_userimpl.cc` file of the class which contains the operation.

The code generator generates an include directive for each specification statement it meets. Thus, for each specification statement the user has to implement a corresponding file with the name: `vdm_<ClassName>_<OperationName>-<No>.cc`,

where `<OperationName>` is the name of the operation in which the specification statement appears, and `<No>` is a sequential numbering of the specification statements that appear in the specific operation.

3.2.3 Implementing the Main Program

We have now implemented the files necessary to compile, link and run the code, except for the main program.

Let us therefore write a main program for the sort example.

First of all, we will start by specifying the main program in VDM++.

```
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     )
```

We will now implement a C++ main program with the same functionality as in the above specified VDM++ method. The main program is implemented in the file `sort_pp.cc` and the complete program is shown in appendix C.8.

The C++ file, containing the main program, should start by including all the necessary header files. These include one header file per VDM++ class, the header of the VDM C++ Library, called `metaiv.h`, and the standard library class `<fstream>` (in order to generate output).

Let us now, step by step, translate the above listed VDM specification to C++. Line `03` and `04` specify two integer lists. Translated to C++, one will get the following code:

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

Line `06` declares an object reference `smach` to an instance of the class `SortMachine`.

The following line implements this in C++:

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

Line `07` declares a variable `res` of type `seq of int`, which will later be used to contain the sorted integer sequences. The C++ code for this is just:

```
type_iL res;
```

Let us now show how to call a specific sorting method.

As can be seen from the VDM++ specification and the generated C++ code, the class `SortMachine` has an object reference to the abstract class `Sorter` as an instance variable. The subclasses of `Sorter` implement the different sorting algorithms. The method `SetAndSort` of the `SortMachine` class takes two parameters: An instance of a subclass of `Sorter` and a sequence of integers. The method sets the mentioned instance variable to refer to the specified subclass and thereby to a specific sorting algorithm, and it afterwards calls the `Sort` method of this class with the integer sequence as parameter. The result will be a sorted integer sequence.

Line `08` declares an object reference `dos` to an instance of the class `DoSort`, and line `08` calls the `SetAndSort` method of the `SortMachine` class with the declared object reference `dos` and the integer sequence `arr1` as arguments. The result is assigned to `res`.

All code generated VDM++ types have an `ascii` method which returns a string containing an ASCII representation of the respective VDM value. This method

is being used here to print relevant log messages to standard output during execution. A reference to the `SortMachine` class can be obtained by calling the `ObjGet_vdm_SortMachine` function defined in the code generated class `CGBase`

```
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";
```

In order to sort `arr2` with the sorting algorithm defined in class `ExplSort`, the following code can be written analogously:

```
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";
```

In order to sort `arr2` with the sorting algorithm implemented in class `ImplSort`, the following code can be written:

```
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";
```

Note, that the interface to the code is independent of having implicit or explicit functions/operations.

One could also imagine that one wants to call the post condition function for `ImplSort`. The Code Generator has generated a function called `vdm_post_ImplSorter` in class `vdm_ImplSort` for it. This function can be called in the ususal way.

```
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";
```

Instead of calling the `SetAndSort` method of the `SortMachine` class, one can choose to first set the desired sorting algoritm by calling `SetSort` and afterwards to call the `GoSorting` method, as shown in line `16` to `18`. We choose here the `MergeSort` algorithm, well-knowing that the resulting C++ code will imply a run-time error. The resulting code is shown in the following:

```
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";
```

The described main program is implemented in the file named `sort_pp.cc` and it is listed in Appendix C.8.

3.2.4 Substituting Parts of the Generated C++ code

Finally, we should say some words about the possibilities of substituting generated C++ code with handwritten code. This can mainly be useful in two situations:

- The user wants to implement code for constructs that are not supported by the Code Generator.
- The user wants to implement some existing components more efficiently.

For the sort example, one could imagine that the user wants to implement a handcoded version of the `MergeSorter` function, as it contains a construct not supported by the Code Generator. The user then has to substitute the code generated function `vdm_MergeSort::vdm_MergeSorter` with a handcoded version. In order to do so, a new function has to be written. This must have the same declaration header as the code generated version found in `MergeSort.cc`:

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

In order to substitute a code generated function with a handwritten function, the function has to be implemented in a file named `MergeSort_userimpl.cc` and the following two defines have to be added to the `MergeSort_userdef.h` file:

```
#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
```

In this way, you can substitute specific functions generated by the Code Generator with handwritten functions.

3.3 Compiling, Linking and Running the C++ code

After the user has handwritten the above described files, he is now in a position to compile, link and run the C++ code.

C++ code generated by this version of the VDM++ to C++ Code Generator must be compiled using one of the following supported compilers:

- GNU gcc version 3, on Sun Solaris 2.6, HP9000/700 running HP-UX 10 and PC's running Linux, or
- VC++ version 6.0 on Windows NT/2000 or Windows 98.

In order to create an executable application, the code must be linked to the following libraries:

- `libCG.a`: Code generator auxiliary functions. This library is released with the VDM++ to C++ Code Generator and is described in Appendix B.
- `libvdm.a`: The VDM C++ Library. This library is released with the VDM++ to C++ Code Generator and is described in [LibMan].
- `libm.a`: The math library corresponding to the compiler.

The `Makefile` used in the implementation of the sort example is listed in appendix D. To compile the main program `sort_pp`, you must type `make sort_pp`.

You can now run the main program `sort_pp`. Its output is listed below. Note that a run-time error has occurred during execution of `MergeSort`. This is caused by the fact that we have tried to execute an unsupported construct. The position information which has been included in the generated code, leads to the origin of the error in the underlying specification.

```
$  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 Unsupported Constructs

In this version of the Code Generator the following VDM++ constructs are not supported:

- Expressions:
  - Lambda.
  - Compose, iterate and equality for functions.

- Function type instantiation expression. However, the code generator supports function type instantiation expression in combination with apply expression, as in the following example:

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

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

- The concurrency part of VDM++, that is the `#act`, `#fin` `#active`, `#waiting` and `#req` expressions.

- Statements:
  - Always, exit, trap and recursive trap statements.
  - Start and start list statements.

- Type binds (see [LangManPP]) in:
  - Let-be-st expression/statements.
  - Sequence, set and map comprehension expressions.
  - Iota and quantified expressions.

  As an example the following expression is supported by the Code Generator:

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

  whereas the following is not (caused by the type bind `n: nat`):

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

- Patterns:
  - Set union pattern.
  - Sequence concatenation pattern.

- Threads

- Synchronization definitions

The Code Generator is able to generate compilable code for specifications including these constructs, but the execution of the code will result in a run-time error if a branch containing an unsupported construct is executed. Consider the following function definition:

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

In this case code for `f` can be generated and compiled. The compiled C++ code corresponding to `f` will result in a run-time error if `f` is applied with the value 2, as type binds in iota expression are not supported.

Note that The Code Generator will give a warning whenever an unsupported construct is encountered.

5 Code Generating VDM Specifications - The Details

This section will give you a detailed description of the way VDM++ constructs are code generated, including classes, types, functions, operations, instance variables, values, expressions and statements.

This description should be studied intensively if you want to use the Code Generator professionally.

Note: This section focuses on the different VDM++ constructs and their mapping to C++ code, NOT on the overall structure of the generated C++ files. The reader is referred to Section 2.4 and Section 3 for a description of the overall structure.

5.1 Code Generating Classes

For each VDM++ class a corresponding C++ class is generated. The inheritance structure of the VDM++ classes corresponds exactly to the inheritance structure

of the generated C++ classes. However, the generated classes are tightly coupled to the VDM C++ Library, as it is the case for all the generated types in the Code Generator. In order to fully understand how the type system works in the generated code you should read the documentation of this library [LibMan]. However, we will give a small introduction to the VDM C++ Library below.

5.1.1 Object References in the VDM C++ Library

The VDM C++ Library is structured with two superclasses: *Common* and *MetaivVal*. For every VDM++ type a corresponding C++ class exists which is a subclass to *Common*, and correspondingly for every kind of value in VDM++ a corresponding C++ class exists which is a subclass to *MetaivVal*. Thus, the VDM C++ Library is structured in a type and a value system, as it is illustrated in Figure 6.

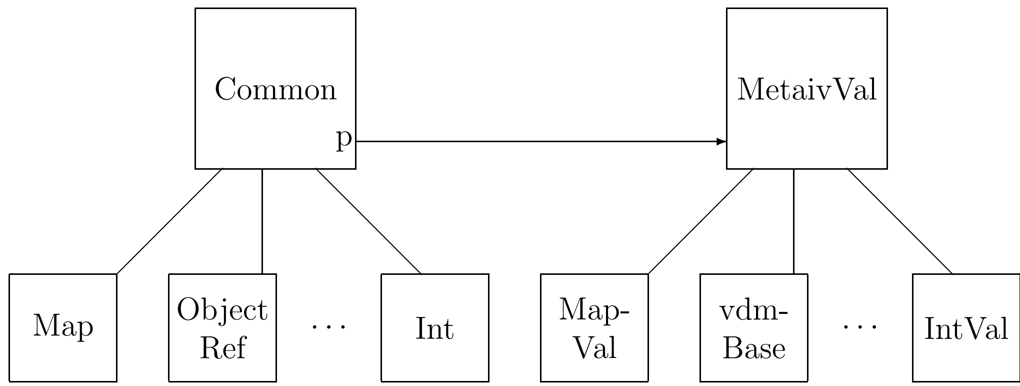

Figure 6: The overall Structure of VDM C++ Library

When an object of a class, say `Int`, is created then an object of *IntVal* is created automatically too, which contains the integer value. In addition the pointer `p` from the `Int` object will be set to point to that `IntVal` object.

Let us now have look at how the VDM++ type "object reference" is reflected in the VDM C++ Library. As for all other types, the VDM C++ Library provides two classes: in the value part system the class `vdmBase` and in the type part system class `ObjectRef`. When creating an instance of an `ObjectRef`, the constructor takes a pointer to an object of the corresponding value part side as input, that is an object of class `vdmBase`. The declaration of one of the typically used constructors of the `ObjectRef` class is listed below:

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

5.1.2 The Inheritance Structure of the Generated Code of Classes

The Code Generator uses the object reference support of the VDM C++ Library in the following way. All the C++ classes corresponding to VDM++ classes inherit from the `vdmBase` class. In addition, for every class in the VDM++ specification a corresponding class is generated that represents the object reference of exactly this VDM++ class. This C++ class inherits from the `ObjectRef` class.

The inheritance structure of the generated C++ class of the sorting example and the VDM C++ Library is shown in Figure 7.

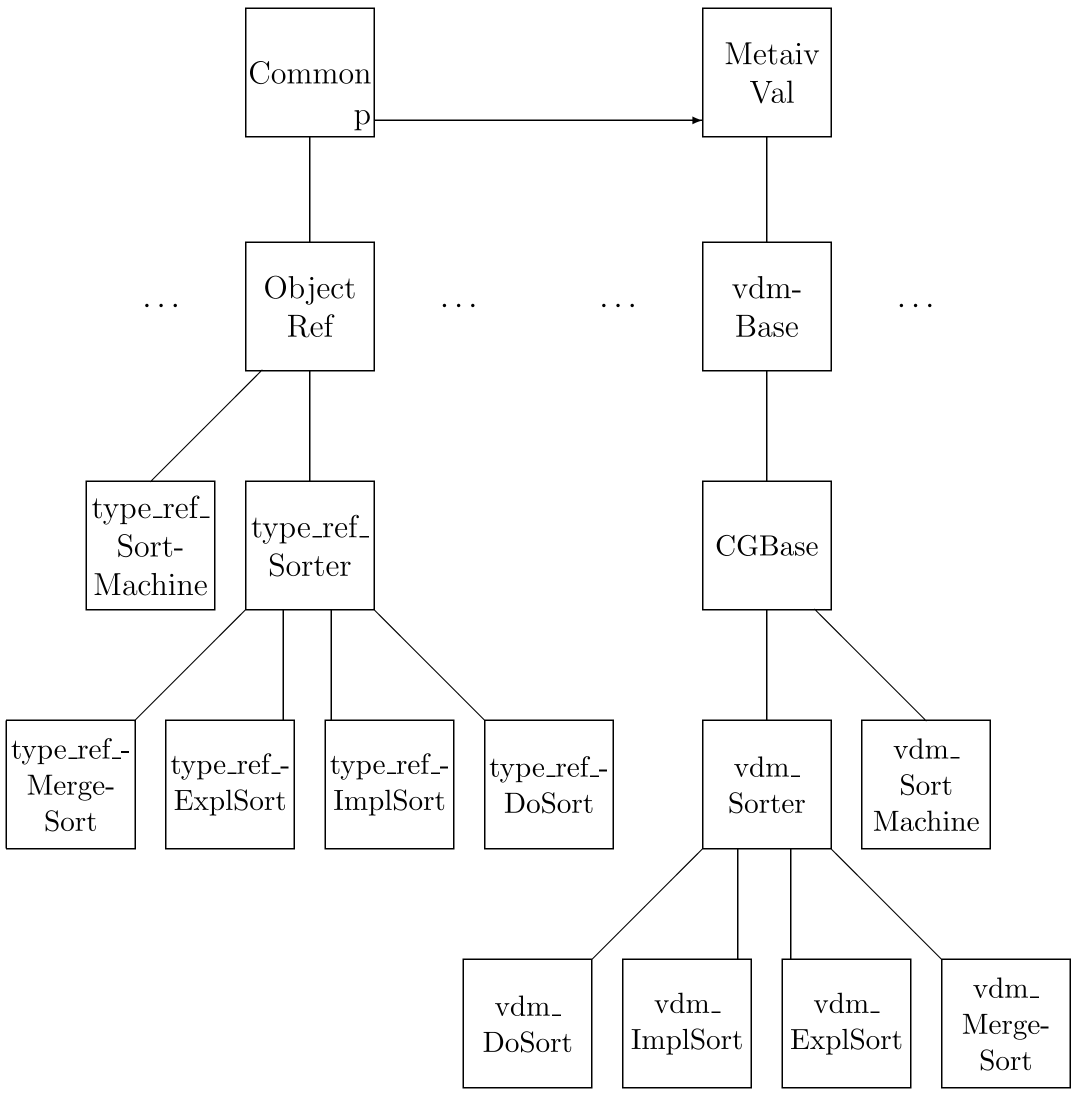

Figure 7: The inheritance structure of the C++ classes of the sorting example and the VDM C++ Library

In the type system part specialised object reference classes are generated type_ref_<Classname> for each VDM++ class. Consider the declaration the class type_ref_DoSort:

```
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"; }
} ;
```

The class contains a constructor that takes a pointer to the `vdmBase` class. Constructing an object reference to an object of class `ExplSort` can be done in the following way:

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

As it can be seen from Figure 7, the generated C++ classes do not inherit directly from the *vdmBase*, but through the class *CGBase*. This class is also generated by the Code Generator.

The `CGBase` class is declared and defined in the files *CGBase.cc* and *CGBase.h*. Apart from the class definition the *CGBase* files also consist of definitions of some extern functions. Altogether, this code provides functions that make it possible to extract the value part (that is the actual C++ object reference) of the object reference type.

For the *Sorting* example consider an extract of the *CGBase.h* file:

```
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
};
```

For each VDM++ class a global function is generated: `ObjGet_vdm_<ClassName>` . The function takes an *ObjectRef* and returns a pointer to the corresponding object of the C++ class. Furthermore, a unique tag is defined for each class description.

An example of constructing an object reference and applying the functions within the class `DoSort` is given below:

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

As the implementation of the VDM C++ Library is based on reference counters, it will delete the pointer to the instance of class `vdm_A` when no existing objects of class `ObjectRef` refer to it. To illustrate this, consider the following example:

```
{
  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.
```

You should **never** directly declare a pointer to a vdm class and instantiate this to an *ObjectRef*. This can go wrong because the VDM C++ Library is based on reference counters, and will delete the objects when there is pointer (at least from the point of view of the VDM C++ Library) that points at the object reference.

```
{
  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 The Structure of a Generated Class

The generated C++ classes contain:

- C++ functions that implement VDM++ functions and operations.
- Some auxiliary functions for object reference gymnastics.
- Constructors/destructors of the class.

The access modifiers in the class follow those specified in the VDM++ class, with the exception of type definitions, for which it is not meaningful to give an access modifier. Thus for example a function which is public at the VDM++ level will be code generated as a public member of the corresponding C++ class, and so on.

Consider the declaration of the C++ `vdm_DoSort` class:

```
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 &);
};
```

The class consists of the VDM++ functions `vdm_DoSorting`, `vdm_InsertSorted` and `vdm_Sort`.

The auxiliary functions are:

- `Get_vdm_DoSort`: returns the reference pointer to the object self.
- `Get_vdm_Self`: returns the object reference pointer to the object self.
- `vdm_GetId`: returns the unique tag of the class.

5.2 Code Generating Types

In Section 3.1 we have already given a short introduction to the way VDM++ types are mapped into C++ code.

Here we will give a more detailed description of this topic.

Section 5.2.1 gives a motivation for the strategy used when code generating VDM++ types. Section 5.2.2 then describes the mapping of each VDM++ type into C++ code. Section 5.2.3 summarizes the used name conventions for types.

5.2.1 Motivation

The type scheme of the Code Generator can be split into two parts:

- The type scheme used in function headers of the generated C++ code.
- The type scheme used in the rest of the generated C++ code.

The type scheme used in function headers uses C++ types that have been code generated. The type scheme used in the rest of the generated code uses the fixed implementation of each VDM++ data type found in the VDM C++ Library. The VDM C++ library (`libvdm.a`) is described in [LibMan].

Let us imagine we have a VDM function with a `seqofchar` as input parameter. Then the corresponding C++ function will take a parameter of type `type_cL` as input. The type `type_cL` is a code generated type, where `c` resembles the VDM type `char`, and `L` resembles the VDM type `seq`. The function implementation

however uses only the type `Sequence` found in the VDM C++ Library instead of the type `type_cL`.

The code generated types obviously improve the generated C++ code. They offer the possibility of catching more type errors at compilation time, and they are more informative for the user.

With the introduction of new types a new problem arise: We have to ensure, that a `seq of char` in class `A` is the same type as a `seq of char` in class `B`.

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

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

In VDM++ the types `A‘C` and `B‘D` are equivalent. This is however not the case in C++, because C++ uses name equivalence except for the basic data types.

The generated code has to ensure two things:

- An anonymous VDM++ type may only be code generated once in order to ensure type correctness in the generated code.
- The generated type name should be readable and understandable.

The first problem is solved by generating `<Class Name>_anonym`files. The `<Class Name>_anonym.h`file contains type declarations for all types that are potentially declared by other classes (anonymous types) as well. The `<ClassName>_anonym.cc` contains the implementation of these types. Moreover, it contains macro defintions for them.

Note, that types which are not anonymous, i.e. composite types and type names, are not declared in the <`ClassName`>`_anonym.h` file, but instead in the `<ClassName>.h`file.

Let us show you the anonymous header file generated for class A in the above listed example.

`A_anonym.h` looks like:

```
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
```

The first `#define` statement defines a macro for type `D` in module `A`. It will be replaced with type `type_cL`. The `TAG_type_cL` ensures a unique tag for the type `type_cL` in the generated code. The two `#ifndef` statements ensure that the `TAG_type_cL` and the `type_cL` are only defined once, either in the file `A_anonym.h` or in the file `B_anonym.h`.

The second problem is solved by the chosen name conventions for generated C++ types. The strategy for generating type names is to unfold types into what we could call a canonical form and then give the canonical form a name based on the type names and type constructors involved. The next two subsections will give you more information about the used notation.

5.2.2 Mapping VDM++ Types to C++

This section describes how VDM types are mapped into C++ types.

- *The Boolean Type*

  The VDM `bool` type is mapped to the VDM C++ library class `Bool` and it is abbreviated with the character `b`.

- *The Numeric Types*

  The VDM `nat`, `nat1` and `int` types are all mapped to the VDM C++ library class `Int` and they are abbreviated with the character `i`. The VDM

real and rat types are mapped to the VDM C++ library class Real and they are abbreviated with the character r.

- *The Character Type*

  The VDM type char is mapped to the VDM C++ library class Char and it is abbreviated with the character c.

- *The Quote Type*

  The VDM Quote type is mapped to the VDM C++ library class Quote and it is abbreviated with the character Q. As for all types, a unique tag has to be ensured for quotes. In the following we will show how the quote <Hello> is code generated.

  The following code will be added in the file <ClassName>_anonym.h:

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

  The following code will be added in the file <Clasname>_anonym.cc:

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

  The declared quote value may now be referenced as quote_Hello in the C++ code.

- *The Token Type*

  The token type is implemented using the C++ class Record. However, the tag of token records is always equal to TOKEN, which is a macro declared in the file cg_aux.h (see Appendix B), and the number of fields in token records is always equal to 1. The VDM++ value mk_token(<HELLO>) can e.g. be constructed in the following way:

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

- *The Sequence Type*

  To handle the compound types `set`, `sequence` and `map`, templates are introduced. These templates are also defined in the VDM C++ library and they are based on the C++ classes `Set`, `Sequence` and `Map` in the C++ VDM library.

  The `seq` type is abbreviated with the character `L`. As an example let us show how the VDM type `seq of int` is code generated:

  ```
  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"; }
  } ;
  ```

  The VDM `seq` type is mapped into a class that inherits from the template `SEQ` class. In case of `seq of int`, the argument of the template class is `Int`, the C++ class representing the basic VDM type `int`. The name of the new class is made in the following way:

  `type` : signals an anonymous type
  `i`: signals int
  `L`: signals sequence

  Note also that several constructors for the `type_iL` class have been generated together with a `GetTypeName` function.

- *The Set Type*

  The VDM `set` type is handled in the same way as the VDM `seq` type. The template class `SET` is used rather than `SEQ` and the type is abbreviated with the character `S`.

- *The Map Type*

  The VDM `map` type is handled in the same way as the VDM `seq` type. The template class `MAP` is used rather than `SEQ` and the `Map` template class takes two arguments rather than one. The `map` type is abbreviated with the character `M`.

- *The Composite/Record Type*

Each composite type is mapped into a class that is a subclass of the VDM C++ library `Record` class. For example, the following composite type defined in a class `M`

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

will be code generated as:

```
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);
} ;
```

As you can see, a record named `A` in a class `M` will be given the name: `TYPE_M_A`.

Several member functions have been added to the generated C++ class definition:

- Two constructors have been added.
- The function `GetTypeName` has been added.
- An initialisation function `Init` has been added. This function initialises the record fields to the corresponding values of the input parameters and returns a reference to the object.
- For each field in the record, two member functions have been added in order to get and set its value. The names of these functions match the names of the corresponding VDM record field selectors. If a field selector is missing, the position of the element in the record will be used instead, e.g. `get_1`.

The implementation of the `Init` function and the `set/get` functions can be found in the implementation file of the class, where the record type has been defined. For the above defined record type, the following code can be found in the file `M.cc`:

```
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); }
```

- *The Tuple/Product Type*

  The strategy for handling tuples is very similar to that of composite types. Each tuple type is mapped into a class that is a subclass of the VDM C++ library `Tuple` class. For example, the following tuple:

  ```
  int * real
  ```

  will be code generated as:

  ```
  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);
  } ;
  ```

  The name of the new class is made in the following way:

  `type` : signals anonymous type
  `i`: signals int
  `r`: signals real
  2P: signals tuple with two subtypes/elements

There is however one difference between the code generation of composite types and tuple types. The VDM++ tuple type is an anonymous type. Therefore, the C++ type definition is found in the `<ClassName>_anonym.h` file, not in the `<Classname>.h` file. Likewise, the implementation of the member functions is found in the `<Classname>_anonym.cc`, not in the `<Classname>.cc` file.

- *The Union Type*

  The union type is mapped into the VDM C++ library `Generic` class.

- *The Optional Type*

  The optional type is mapped into the VDM C++ library `Generic` class.

  Note, `nil` is a special VDM++ value (not a type).

- *The Object Reference Type*

  In Section 5.1 it has been described how two C++ classes are generated for each VDM++ class. One of these represents the class itself and the other one is used for handling a reference to the VDM++ class.

  Look at the following example:

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

class N
...
end N
```

  Class M defines a type `A` that is a reference to an object of class `N`. When code generating this example, five classes will be defined: `vdm_M`, `vdm_N`, `type_ref_M`, `type_ref_N` and `type_1NRL`. The last one represents the defined type `seqofN`. The name of the new class is made in the following way:

  `type` : signals anonymous type
  `1`: signals numbers of characters in the name of the class
  `N`: signals class N
  `R`: signals object reference
  `L`: signals sequence

Moreover, the following macro will be defined in file `M_anonym.h`:

5.2.3 Code Generating VDM++ Type Names

The type system of VDM++ and C++ differs as C++ uses name equivalence and VDM++ uses structural equivalence. In VDM++

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

type A and B are equivalent because they are structural equal. However, the corresponding example in C++ is not equivalent because the name of A and B are different.

The Code Generator solves this problem by generating equal names for structural equal types. Thus, the corresponding generated C++ code is (in essence):

```
class type_iL

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

#define TYPE_ClassName_A type_iL
#define TYPE_ClassName_B type_iL
```

Thus all type name definitions are defined through #define directives to a name reflecting the structural content of the type definition.

A generated type name is prefixed with `TYPE` followed by the class name, where the type is defined and finally the chosen VDM name is concatenated.

All anonymous types, i.e. types that are not given a name in the VDM++ specification are prefixed with `type` and a constructed name that reflects the structure of the type. The type name is based on an unfolding of the VDM type and the use of a reverse polish notation.

The table below sketches the naming convention. The names of the VDM types and type constructors are in the first column. In the second row the scheme for generating names corresponding to the VDM type is listed. In the second column *<tp>'s* should be replaced by generated type names for the corresponding VDM type. E.g. the VDM type `map char to int` is given the name `ciM` as `char` translates to `c`, `int` translates to `i` and the map type constructor takes the two argument types and combines with what we could see as a reverse polish operator `M`, giving `ciM`. The naming conventions will be described further in the rest of this note.

VDM	translation	examples
bool	`b`	`b`
nat1 nat int	`i` `i` `i`	`i` `i` `i`
real rat	`r` `r`	`r` `r`
char	`c`	`c`
quote	`Q`	`<Hello>` translates to `Q`
token	`T`	`token` translates to `T`
set	`<tp>S`	`set of char` translate to `cS`
sequence	`<tp>L`	`sequence of real` translates to `rL`
map	`<tp1><tp2>M`	`map set of int to char` translates to `iScM`
product	`<tp1>..<tpn><n>P`	`int * char * sequence of real` translates to `icrS3P`
composite	`<length><name>C`	the composite type `Comp` translates to `4CompC`. Notice that `<length>` is the number of characters in the name of the composite type.
union	`U`	`int | char | real` translates to `U`
optional	`<tp>O`	`[ int ]` translates to `iO`
object ref	`<length><name>R`	a reference to an object of class `Cl` translates to `2ClR`. Notice that `<length>` is the number of characters in the name of the objects class.

VDM	translation	examples
recursive type	`F`	the type T defined as `T = map int to T` translates to `iFM`, i.e. the first unfolding of the type. A recursive type will always be given the name F in a generated type name. An exception to this is a recursive composite type which will get the name as described above. See the following section for more details

In the table above, the `<length>` part used for handling composite and object references is used in order to ensure that type names may be read without ambiguity.

The unfolding of types into a canonical representation is made difficult by the existence of recursive types. Therefore the name of a recursive type will be represented by the name `F`. For example the type `A` in `A = map int to A` will be represented by the type name `iFM`. The type generated for `B` in the following type definition `B = sequence of A` will be `FL`.

Composite types will not be unfolded but are represented by their name, e.g. `Comp :: ..` is represented by the type `4CompC`.

5.2.4 Invariants

When an invariant is used to restrict a type definition in the specification, an invariant function is also available. This invariant function can be called in the same scope as its associated type definition (see [LangManPP]). The VDM++ to C++ Code Generator generates C++ function definitions corresponding to invariants. As an example, consider the following VDM++ type definition in class `M`:

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

The function declaration corresponding to the VDM++ function `inv_S` is listed below. This declaration is placed in the protected part of the C++ class `vdm_M` in file `M.h`:

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

The implementation of this invariant function is found in file `M.cc`.

Note that the VDM++ to C++ Code Generator does not support dynamic check of invariants, and invariant functions must therefore be called explicitly.

5.3 Code Generating Function and Operation Definitions

In VDM++, functions and operations can be defined either explicitly or implicitly. The VDM++ to C++ Code Generator generates C++ function declarations of both implicit and explicit function and operation definitions. These declarations can be found in the `<ClassName>.h` file.

Let us show two examples of generated C++ function declarations:

The operation definition `Sort` in the VDM++ class `ExplSort` is explicit and leads to the following C++ function declaration in class `vdm_ExplSort` in file `ExplSort.h`:

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

The function defintion `ImplSorter` in the VDM++ class `ImplSort` is implicit and leads to the following C++ function declaration in class `vdm_ImplSort` in file `ImplSort.h`:

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

Note: The C++ function declarations are declared as virtual and public in order to correspond to the VDM++ semantic. In VDM++ all functions and operations are virtual and public, in C++ this is only the case, when they are declared as such.

Let us show another example of a function declaration. Look at the function `f` in the following VDM++ specification.

```
class M
  types
```

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

The following function header will be generated for the function f:

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

Note, that type names are used in the function signature rather than the name of the corresponding unfolded type. This is as close to the VDM specification as we can get it. The same strategy will be used in variable declarations.

Explicit Function and Operation Definitions

The VDM++ to C++ Code Generator generates C++ function definitions for explicit VDM++ function and operation definitions. These function definitions are placed in the corresponding `<ClassName>.cc` file. For the above mentioned explicit operation `Sort` in the VDM++ class `ExplSort`, the following C++ function definition is added to the file `ExplSort.cc`:

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

The shown examples give you an idea about how function or operation names are generated: A function or operation name `f` in a class M in a VDM specification will be given the name: `vdm_M::vdm_f`.

Implicit Function and Operation Definitions

Obviously, for implicit function or operation defintions, no C++ function defintion is added to the generated C++ code. Instead, the Code Generator generates

an include preprocessor in the implementation file. For the class `ImplSort` which contains an implicit function, as described, the preprocessor below will appear in the implementation file `ImplSort.cc`:

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

It is then the user's responsibility to implement the implicit function of class `ImplSort` in the file `ImplSort_userimpl.cc`. Note that a compile time error will occur if this file is not created and that the linker will object if an implicit function is not implemented. See Section 3.2.2 for information about implementing implicit functions and operations.

Pre and Post Conditions

Post conditions on operation specifications are ignored by the VDM++ to C++ Code Generator. However, when pre and post conditions are specified for functions, corresponding pre and post functions are available (see [LangManPP]). For each of these functions, a C++ function declaration and a C++ function definition will be generated. The pre and post functions are members of the generated C++ class with access modifier given by that of the corresponding VDM++ function.

Let us show the C++ code which is generated for the post condition of function `ImplSorter` in class `ImplSort`:

The following function declaration is found in file `ImplSort.h`:

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

The implementation of the function can be found in file `ImplSort.cc`.

Runtime checks of pre and post conditions can be optionally generated (either by selecting the corresponding check box in the graphical user interface, or by specifying the `-P` option on the command line). For instance, consider the function `RestSeq` from the `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;
```

When generate with runtime checking of pre and post conditions, the following code is produced:

```
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;
}
```

In this way, assertions at the VDM++ level can be evaluated in the generated code.

Substituting generated C++ functions with handwritten function code

It should be mentioned, that it is possible to substitute generated C++ functions with handwritten C++ code. Section 3.2.4 describes the situations where this is useful and the steps that the user has to perform in order to interface the handwritten code with the generated code.

5.4 Code Generating Instance Variables

The code generation of instance variables is very straightforward. Instance variables are translated into member variables of the corresponding C++ class. These member variables are placed in the protected part of the generated class definition.

Consider the following instance variable declaration in VDM++:

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

The corresponding member declarations generated by the Code Generator in file `A.h` will become:

```
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
};
```

The implementation of the constructor function for class `A` can be found in file `A.cc`. It initializes the instance variables as shown below:

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

Note: Invariant definitions specified in instance variable blocks are ignored by the Code Generator.

Moreover, you probably wonder why the Code Generator generates the type `Sequence`, instead of the type `type_cL`. As it has been mentioned in Section 5.2, the code generated types are only used in the interface to the user. For internal use however, the Code Generator uses the fixed implementation of each VDM++ data type found in the VDM C++ Library.

5.5 Code Generating Value Definitions

Let us now explain the code generated for the definition of constant values.

VDM++ value definitions are translated to static member variables of the generated C++ class. The initialisation of the value variables is done by the generated C++ `Init_<ClassName>` class which is instantiated in the ".cc" file.

Consider the example below:

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

The generated header file `A.h` will look like:

```
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
```

The implementation file will look like:

```
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 Code Generating Expressions and Statements

VDM++ expressions and statements are code generated, such that the generated code behaves like it is expected from the specification.

The undefined expression and the error statement are translated into a call of the function `RunTime` (see Appendix B). This call terminates the execution and reports that an undefined expression was executed.

5.7 Name Conventions

A variable in the specification will be translated to a variable in the generated C++ code. The naming strategy used by the VDM++ to C++ Code Generator is to rename all these variables to: `vdm_<name>`, where <name> is the name appearing in the specification. The function `f` will e.g. be named `vdm_f`. In addition the following names are used by the Code Generator:

- `length_`*record*: A static variable defining the number of fields in the record *record*.
- `pos_`*record*_*field*: A static variable defining the position/index (an integer) of the field selector *field* in the record *record*.
- *name_number*: A temporary variable used by the generated C++ code. The specification/topology statements are numbered in the order in which they are defined in the method, starting by one.

Underscores (‘`_`’) and single quotes (‘`'`’) appearing in variables in the specification will be exchanged with underscore-u (‘`_u`’) and underscore-q (‘`_q`’), respectively, in the generated C++ code.

5.8 Standard Library

Math Library

If a specification using the Math library (the `math.vpp` file) is code generated the functions of the library must be implemented in a file named `MATH_userimpl.cc` as these functions are implicitly defined. A default implementation of this file exists in the directory `vpphome/cg/include`.

IO Library

If a specification using the IO library (the `io.vpp` file) is code generated the functions of the library must be implemented in a file named `IO_userimpl.cc` as these functions are implicitly defined. A default implementation of this file exists in the directory `vpphome/cg/include`.

If use is made of the `freadval` function in the IO library, the class initialiser function is extended (see Section 5.5 for details of initialiser functions). `freadval` is used to read a VDM value from a file. In order to behave correctly on files containing record values, the function `AddRecordTag` is used in the initialiser function to establish the correct relationship between textual record tag names, and the integer tag values used within the generated code. `AddRecordTag` is provided as part of the `libCG.a` library (See Appendix B). For example, suppose that class `M` defines a record type `A`. In the function `init_M` the following line would appear:

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

In this way, when a value of type `M‘A` is read from a file, it will be translated into a record value with the correct tag.

A References

[InstallPPMan] The VDM Tool Group. *VDM++ Installation Guide*. Technical Report, IFAD, October 2000.

[LangManPP] The VDM Tool Group. *The IFAD VDM++ Language*. Technical Report, IFAD, April 2001. ftp://ftp.ifad.dk /pub/vdmtools/doc/langmanpp_letter.pdf.

[LibMan] The VDM Tool Group. *The VDM C++ Library*. Technical Report, IFAD, October 2000.

[SortEx] The VDM Tool Group. *VDM++ Sorting Algorithms*. October 2000. Available in both postscript and RTF formats.

[UserManPP] The VDM Tool Group. *VDM++ Toolbox User Manual*. Technical Report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc /usermanpp_letter.pdf.

B The libCG.a Library

The library, `libCG.a`, is a library of fixed definitions which is used by the generated code. The interface to `libCG.a` is defined in `cg.h` and `cg_aux.h`.

B.1 cg.h

The functions `RunTime` and `NotSupported` are called when a run-time error occurs or when a branch containing an unsupported construct is executed. Both these functions will print an error message and the program will exit (`exit(1)`). If position information is available at the time when one of these functions are

called, the last recorded position in the VDM++ source specification will be printed. This is the case when the code has been generated using the run-time position information option.

The functions `PushPosInfo`, `PopPosInfo`, `PushFile` and `PopFile` are used by the generated code to maintain the position information stack (if run-time position information has been included in the generated code).

`ParseVDMValue` is intended for use by hand implementations of the IO standard library. It takes the name of a file, and a `Generic` reference, and reads the VDM value from the given file. This value is placed in the given reference. The function returns true or false, according to whether it was successful or not.

```
/***
*  * WHAT
*  *    Code generator auxiliary functions
*  * ID
*  *    $Id: cg.h,v 1.15 2001/06/12 15:04:34 paulm Exp $
*  * PROJECT
*  *    Toolbox
*  * COPYRIGHT
*  *    (C) 1994 IFAD, Denmark
***/

#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
```

B.2 cg_aux.h

The definitions in `cg_aux.h` contain auxiliary definitions which are dependent of the library used to implement the VDM++ data types[4].

The functions `Permute`, `Sort`, `Sortnls`, `Sortseq`, `IsInteger` and `GenAllComb` are used by the generated code corresponding to different types of expressions.

```
/***
*  * WHAT
*  *    Code generator auxiliary functions which are
*  *    dependent of the VDM C++ Library (libvdm.a)
*  * ID
*  *    $Id: cg_aux.h,v 1.13 1998/10/28 13:42:58 hanne Exp $
*  * PROJECT
*  *    Toolbox
*  * COPYRIGHT
*  *    (C) 1994 IFAD, Denmark
***/

#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&);
```

[4] In this version of the VDM++ to C++ Code Generator it is only possible to use the VDM C++ Library.

```
Set GenAllComb(const Sequence&);

#endif
```

C Handcoded C++ Files

C.1 DoSort_userdef.h

```
#define TAG_DoSort 200
```

C.2 ExplSort_userdef.h

```
#define TAG_ExplSort 400
```

C.3 ImplSort_userdef.h

```
#define TAG_ImplSort 300
```

C.4 MergeSort_userdef.h

```
#define TAG_MergeSort 100
```

C.5 SortMachine_userdef.h

```
#define TAG_SortMachine 4500
```

C.6 Sorter_userdef.h

```
#define TAG_Sorter 4600
```

C.7 ImplSort_userimpl.cc

We have chosen to implement `vdm_ImplSort::vdm_ImplSorter` as a handwritten version of `MergeSort`.

```
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.Length() == 0)
    return l2;
  else if (l2.Length() == 0)
    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()));
  }
}
```

C.8 sort_pp.cc

```
/***
*  * WHAT
*  *    Main C++ program for the VDM++ sort example
*  * ID
*  *    $Id: sort_pp.cc,v 1.10 2000/10/23 12:14:08 paulm Exp $
*  * PROJECT
*  *    Toolbox
*  * COPYRIGHT
*  *    (C) 1994 IFAD, Denmark
***/

#if defined(_MSC_VER)
#include <fstream>
#else
#include <fstream.h>
#endif
#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()
{
  //  let arr1 = [3,5,2,23,1,42,98,31],
  //      arr2 = [3,1,2]:
  type_iL arr1, arr2;
  arr1.ImpAppend ((Int)3);
  arr1.ImpAppend ((Int)5);
  arr1.ImpAppend ((Int)2);
  arr1.ImpAppend ((Int)23);
  arr1.ImpAppend ((Int)1);
  arr1.ImpAppend ((Int)42);
  arr1.ImpAppend ((Int)98);
  arr1.ImpAppend ((Int)31);
```

```
arr2.ImpAppend ((Int)3).ImpAppend ((Int)1).ImpAppend ((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);

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";

//     def expls  : Sorter := new ExplSort() in
//     res = smach.SetAndSort(expls,arr2);
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";

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

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";

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";

//     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);
  cout << "Evaluating MergeSort(" << arr2.ascii () << "):\n";
  res = ObjGet_vdm_SortMachine(smach)->vdm_GoSorting(arr2);
  cout << res.ascii() << "\n\n";
  return 0;
}
```

D Makefiles

D.1 Makefile for Unix Platform

```
# WHAT
#    Makefile for the code generated VDM++ sort example.
#    (C) 1994, IFAD, Denmark
# ID
#    $Id: Makefile,v 1.14 2000/10/23 12:13:41 paulm Exp $
# PROJECT
#    Toolbox
# COPYRIGHT
#    (C) 1994 IFAD, Denmark
#
# REMEMBER to change the variable TBDIR to fit your directory structure.
#

CCPATH = /opt/gcc-2.95.2/bin/
CC      = $(CCPATH)gcc
CCC     = $(CCPATH)g++
GCC     = $(CC)
CXX     = $(CCC)
TBDIR   = /opt/toolbox
VPPDE   = $(TBDIR)/bin/vppde

INCL    = -I$(TBDIR)/include
LIB     = -L$(TBDIR)/lib -lvdm -lCG -lm

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

all: sort_pp

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

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)

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

$(GENCCFILES): $(SPECFILES)
$(VPPDE) -c -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 sort.ps sort.dvi
rm -f $(SPECFILES:%=%.tex)
rm -f $(SPECFILES:%=%.aux)
rm -f $(GENFILES)
rm -f $(GENCCFILES)
```

D.2 Makefile for Windows Platform

```
# WHAT
#    Windows NT nmake makefile for the VDM++ sort example.
#    (for VC++ 5.0)
#    (C) 1994, IFAD, Denmark
# ID
#    $Id: Makefile.winnt,v 1.11 2000/10/23 12:13:41 paulm Exp $
# PROJECT
#    Toolbox
# COPYRIGHT
#    (C) 1994 IFAD, Denmark
```

```
TBDIR=g:/Program Files/The IFAD VDM++ Toolbox v6.6


CPP = cl
CPPFLAGS = -nologo -c -GX -MT -I"$(TBDIR)/cg/include"
LDFLAGS  = "$(TBDIR)/cg/lib/CG.lib" "$(TBDIR)/cg/lib/vdm.lib"
VPPDE = "$(TBDIR)"/bin/vppde

COMPILE=$(CPP) $(CPPFLAGS)

.SUFFIXES: .cpp .obj .exe
.cpp.obj:
$(COMPILE) -Tp $<

.obj.exe:
$(CPP) $(LDFLAGS) $^

all: sort_pp.exe

GENCCFILES = 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

OBJS = DoSort.obj ExplSort.obj ImplSort.obj MergeSort.obj \
       SortMachine.obj Sorter.obj CGBase.obj

sort_pp.exe: sort_pp.obj $(OBJS)

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

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

##############################################################
#### 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 *.obj sort_pp.exe
rm -f $(GENFILES)
rm -f $(GENCCFILES)
```