VDMTools

The VDM-SL to C++ Code Generator

ver.1.0

**How to contact:**

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

The VDM-SL to C++ Code Generator 1.0
    — Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

Contents

1 Introduction

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

The Code Generator is an add-on feature to the VDM-SL Toolbox. Its installation is described in the document [3]. The following text is an extension to the *VDMTools User Manual (VDM-SL)* [6] and it gives you an introduction to the VDM-SL to C++ Code Generator.

The Code Generator supports approximately 95% of all the VDM-SL 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-SL specification has to satisfy in order to generate correct C++ code. Moreover, this section describes how to invoke the VDM-SL to C++ Code Generator from the VDM-SL 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-SL 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-SL and C++ data types, and it describes some of the design decisions made, when developing the VDM-SL 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-SL 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 [5]. 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-SL 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-SL 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 modules of the VDM-SL specification are syntax checked in order to generate correct code.

Moreover, the Code Generator can only generate code for modules, which are type correct.[1] If a module has not been type checked before 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-SL Toolbox.

The VDM-SL Toolbox is started with the command `vdmgde`. In order to generate code corresponding to the sort example, create a new project containing the file `sort.rtf` which can be found in the directory `/vdmhome/examples/sort`. See [6] 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 the module *DefaultMod* 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 module - in this example only one - in the specification

---

[1]There exist two classes of well-formedness as explained in [4]. In the current context we mean possible well-formed type correctness.
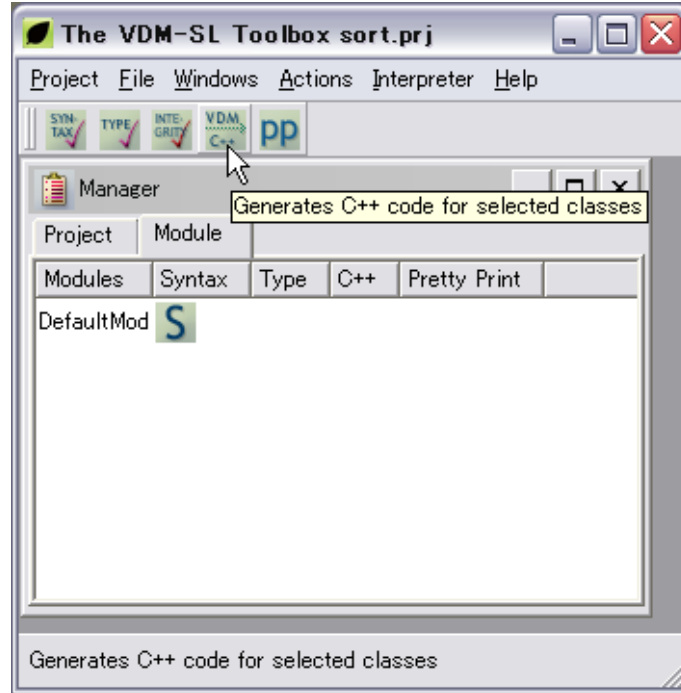
Figure 1: Code Generating the Sort Example

and the Toolbox signals this 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-SL 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-SL specification which causes the run-time error. This feature can be set in the option menu, as shown in Figure 4.
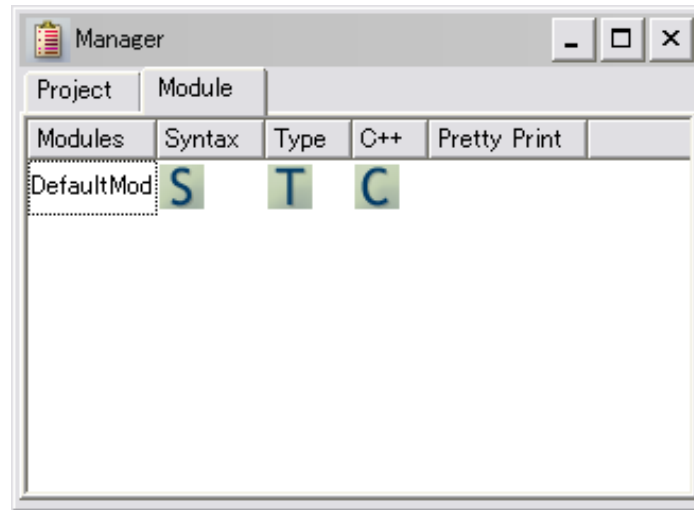
Figure 2: Code Generating the Sort Example
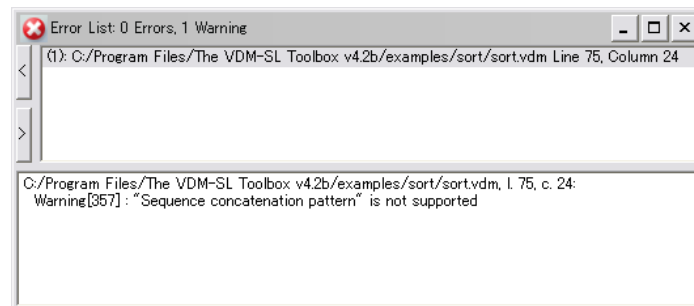
Figure 3: A Warning Generated by the Code Generator

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 appear:

```
Last recorded position:
In: sort.vdm. At line: 43 column: 18
The construct is not supported: Sequence concatenation pattern
```
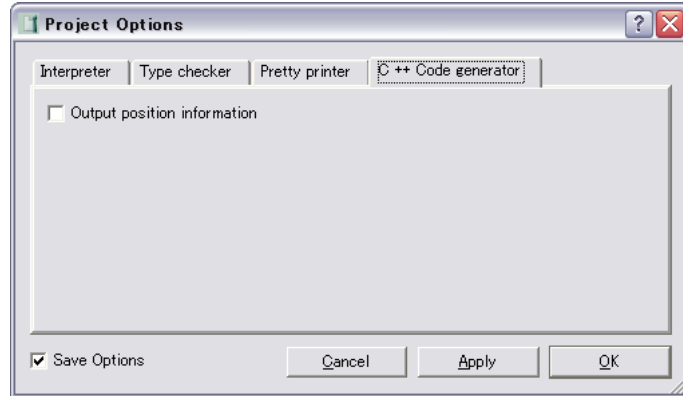
Figure 4: Option for Generating Position Information in Run-time Errors

2.3   Using the Command Line Interface

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

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

```
vdmde -c [-r] specfile, ...
```

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

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

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.

2.4   Generated C++ Files

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

For each module, four files are generated:

- <ModuleName>.h

- <ModuleName>.cc

- <ModuleName>_anonym.h

- <ModuleName>_anonym.cc

The `<ModuleName>.h` file contains C++ function declarations corresponding to all the functions and operations defined in the specific VDM module. Moreover it contains the class definitions corresponding to the composite types defined in the module.

The `<ModuleName>.cc` file contains the implementation of the functions and operations defined in the specific VDM module. In addition, for every class generated for a record type, the implementation of its member functions is found here.

The purpose of the `<ModuleName>_anonym.h` and `<ModuleName>_anonym.cc` files is to declare and implement all the anonymous types that appear in the specific module.

The VDM-SL to C++ Code Generator supports both code generation of structured and code generation of flat VDM-SL specifications (see [4]). However, a flat specification is transformed into a default module named `DefaultMod`. This module exports all VDM-SL cconstructs (except the module state).

The code corresponding to each VDM-SL module is divided into a header and an implementation file. Both files will be named with the module 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].

---

[2]On the Windows 2000/XP/Vista platform this can be set in the `Registry`

3   Interfacing the Generated Code

We have now reached the point, where a number of C++ files have been generated from a VDM-SL 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-SL constructs, especially VDM-SL 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-SL Types - The Basics

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

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

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

defined in module `DefaultMod` is for example code generated as follows:

```
class type_rL : public SEQ<Real>{
...
};

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

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 [1].

- *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 module `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.

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-SL 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 module name, where the type is defined, and finally the chosen VDM name is concatenated.

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

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

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 module M in a VDM specification will be given the name: `vdm_M_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

---

[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.

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-SL 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-SL module and C++ files per VDM-SL specification.
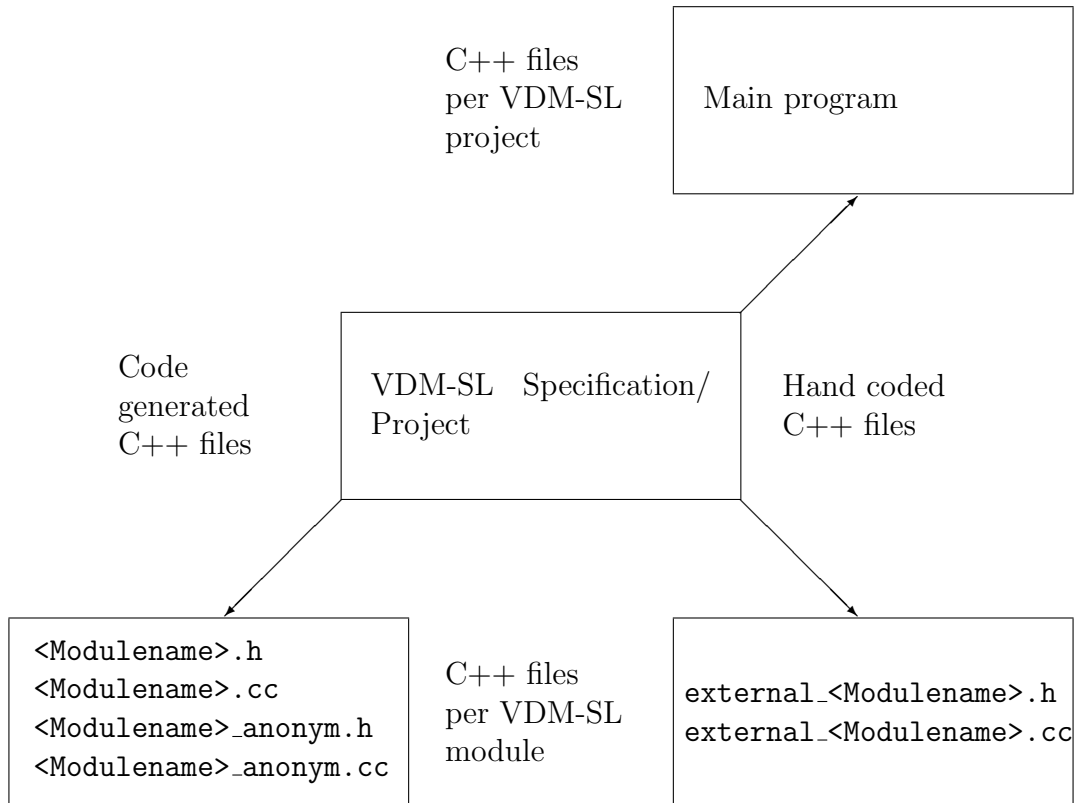

Figure 5: C++ Files per VDM-SL Project

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 module define offsets for record tags.

2. Implement implicit functions/operations and specification statements contained in the VDM-SL 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.

3.2.1   Definition of Offsets for Record Tags

A composite type (record type) consists in the VDM-SL 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 module 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 module ensures that the tags are unique.

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

For the sort example this implies, that the user has to implement one file, namely `DefaultMod_userdef.h`. This file could possibly contain the following definition.

```
 #define TAG_DefaultMod 100
```

3.2.2   Implementing Implicit Functions/Operations and Specification Statements

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

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

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

The function `ImplSort` in module `DefaultMod` is given the name `vdm_Default-Mod_ImplSort`.

One possible implementation of this function is listed in Appendix B.2.

Thus, the user has to write a C++ function definition for the operation and add it to the `<ModuleName>_userimpl.cc` file of the module 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_<ModuleName>_<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-SL.

```
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))
```

We will now implement a C++ main program with the same functionality as in the above specified VDM-SL method. The main program is implemented in the file `sort_ex.cc` and the complete program is shown in appendix B.3.

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

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

Let us now, step by step, translate the above listed VDM specification to C++. Line `03` specifies a list of reals. Translated to C++, one will get the following code:

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

Line `04` declares a variable `res` of type `seq of real`, which will later be used to contain the sorted sequence. Line `05` declares a variable of type `bool`. The C++ code for this is just:

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

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

Line `06` calls the `DoSort` function with the declared list of reals as argument. The result will be a sorted sequence.

All code generated VDM-SL 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 `cout` (standard output) during execution.

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

In order to sort `l` with the function `ExplSort` or `ImplSort`, the following code can be written analogously:

```
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";
```

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_Default-Mod_post_ImplSort` for it. This function can be called in the ususal way.

```
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";
```

As can be seen from the VDM-SL specification and the generated C++ code, the Code Generator has generated an invariant function for the invariant defined for the type `PosReal`. This function is called `vdm_DefaultMod_inv_PosReal` and it can be called in the ususal way:

```
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";
```

Finally, we can choose to call the `MergeSort` function (line 12), well knowing, that this will imply a run-time error.

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

The described main program is implemented in the file named `sort_ex.cc` and it is listed in appendix B.3.

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 `MergeSort` function, as it contains a construct not supported by the Code Generator. The user then has to substitute the code generated function `vdm_DefaultMod_MergeSort` 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 `DefaultMod.cc`:

```
type_rL vdm_DefaultMod_MergeSort(const type_rL
                                       &vdm_DefaultMod_l) {
...
}
```

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

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

For modules that do not contain an implicit function or operation definition, a `<ModuleName>_userimpl.cc` is optional. If the file is created the user has to add another line to the `<ModuleName>_userdef.h` file in order to specify that a `<ModuleName>_userimpl.cc` file now exists:

```
#define DEF_DefaultMod_USERIMPL
```

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-SL to C++ Code Generator must be compiled using one of the following supported compilers:

- Microsoft Windows 2000/XP/Vista and Microsoft Visual C++ 2005 SP1

- Mac OS X 10.4 or higher

- Linux GNU gcc 3, 4 and Kernel 2.4, 2.6

- Solaris 10

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-SL to C++ Code Generator and is described in Appendix A.

- `libvdm.a`: The VDM C++ Library. This library is released with the VDM-SL to C++ Code Generator and is described in [1].

- `libm.a`: The math library corresponding to the compiler.

The `Makefile` used in the implementation of the sort example is listed in Appendix C. To compile the main program `sort_ex`, you must type `makesort_ex`.

You can now run the main program `sort_ex`. 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_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 Unsupported Constructs

In this version of the Code Generator the following VDM-SL 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}
    ```

- Statements:

  - 'using' in call statement.

  - Always, exit, trap and recursive trap statements.

- Type binds (see [4]) 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.

- Local Function Definitions.

- Higher order function definitions.

- Function Values.

- Parameterized modules.

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-SL constructs are code generated, including modules, types, functions, operations, states, 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-SL 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 Types

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

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

Section 5.1.1 gives a motivation for the strategy used when code generating VDM-SL types. Section 5.1.2 then describes the mapping of each VDM-SL type into C++ code. Section 5.1.3 summarizes the used name conventions for types.

5.1.1   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-SL data type found in the VDM C++ Library. The VDM C++ library (`libvdm.a`) is described in [1].

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 module `A` is the same type as a `seq of char` in module `B`.

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

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

In VDM-SL 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-SL 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 `<ModuleName>anonym` files. The `<ModuleName>_anonym.h` file contains type declarations for all types that are potentially

also declared in other modules (anonymous types). The `<ModuleName>_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 `<ModuleName>_anonym.h` file, but instead in the `<Module Name>.h`file.

Let us show you the anonymous header file generated for module 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.

5.1.2   Mapping VDM-SL 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 A), and the number of fields in token records is always equal to 1. The VDM-SL 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 module `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-SL 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-SL value (not a type).

5.1.3   Code Generating VDM-SL Type Names

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

```
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_ModuleName_A type_iL
#define TYPE_ModuleName_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 module 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-SL 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	i	i
nat	i	i
int	i	i
real	r	r
rat	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

VDM	translation	examples
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.
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`.

5.1.4 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 [2]). The VDM-SL to C++ Code Generator generates C++ function definitions corresponding to invariants. As an example, consider the following VDM-SL type definition in class `M`:

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

The prototype corresponding to the VDM-SL function `inv_S` is listed below. It is assumed that `S` is defined in module A:

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

Declarations (prototypes) of invariant functions will always be placed in the header file, as all types currently are exported.

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

5.2 Code Generating Function and Operation Definitions

The VDM-SL to C++ Code Generator generates C++ prototypes of both implicit and explicit functions and operations. If a function[4] is exported from its defining module, the corresponding C++ prototype will be placed in the header file. Functions which are not exported are declared as `static` functions in the implementation file.

Explicit Function and Operation Definitions

The VDM-SL to C++ Code Generator generates C++ function definitions for all explicit VDM-SL functions and operations. These function definitions are placed in the implementation file.

---

[4]In the following, function refers to both function and operation.

Implicit Function and Operation Definitions

In addition to the C++ function prototypes corresponding to implicit functions, the Code Generator generates an include preprocessor in the implementation file. If the module A e.g. contains some implicit functions, then the preprocessor below will appear in the implementation file of module A (`A.cc`):

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

It is then the user's responsibility to implement the implicit functions in the file `A_userimpl.cc`. Note that a compile time error will occur if this file is not created.

Pre and Post Conditions

When pre and post conditions are specified, corresponding pre and post functions are implicitly defined (see [4]). To each of these functions, a C++ prototype and a C++ function definition will be generated. The pre and post functions will implicitly be exported and hence placed in the header file if their corresponding VDM-SL function definition is exported.

Note that the current version of the VDM-SL to C++ Code Generator does not support dynamic checking of pre and post conditions. In order to have these checked, the pre and post functions must explicitly be called.

5.3   Code Generating Value and State Definitions

The code corresponding to the state and value definitions is straightforward. To each value and each state component a C++ variable is declared. All state components and values that are not exported are declared as `static` variables in the implementation file. For each exported value, an `extern` variable declaration is placed in the header file.

In addition, the state definition is treated as a record type definition (see Section 5.1.2).

The Initialisation Function

Initialisation of values and state components is done by a function named `init_`*module*. In order to code generate an initialisation predicate, the predicate must be in the form required by the interpreter (see [4]). If this is not the case, the code generator generates an "include statement", including a user defined definition of the initialisation. Consider the following the state definition of module `A`:

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

In this case, the initialisation predicate is in the form required by the interpreter and the initialisation of the state will be fully generated. If the initialisation predicate was not in this form, the code generator would generate the following include directive inside `init_A`

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

It is the user's responsibility to ensure that this file exists.

5.4   Code Generating Value Definitions

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

The code corresponding to the state and value definitions is straightforward. To each value and each state component a C++ variable is declared. All state components and values that are not exported are declared as `static` variables in the implementation file. For each exported value, an `extern` variable declaration is placed in the header file.

Consider the specification of module `A`:

```
module A
```

```
exports
  values b: int

definitions

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

The corresponding *A.cc* file is listed below:

```
#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.

}
```

Note that `vdm_A_a` and `vdm_A_c` are static variables, wheras the exported value `vdm_A_b` is not.

The corresponding header file of the `A` module (`A.h`) is listed below:

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

Note that `vdm_A_b` is declared as an `extern` variable

The function *init_A()* should be called by the user in the main program.

5.5   Code Generating Expressions and Statements

VDM-SL 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 A). This call terminates the execution and reports that an undefined expression was executed.

5.6   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-SL to C++ Code Generator is to rename all these variables to: **vdm_***module***_***name*, where *module* is the name of the module in which *name* is defined. If the function `f` e.g. is defined in module M, then the C++ function corresponding to `f` will be named **vdm_M_f**. In addition the following names are used by the Code Generator:

- **init_***module*: A function initialising values and the state of module *module*.

- **length_***module***_***record*: A `static` variable defining the number of fields in the record *record*, defined in module *module*.

- **pos_***module***_***record***_***field*: A `static` variable defining the position/index (an integer) of the field selector *field* in the record *record*, defined in module *module*.

- *name***_***number*: A temporary variable used by the generated C++ code.

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.

5.7   Standard Library

Math Library

If a specification using the Math library (the `math.vdm` respectively `mathflat.vdm` 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 `vdmhome/cg/include`.

IO Library

If a specification using the IO library (the `io.vdm` respectively `ioflat.vdm` 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 `vdmhome/cg/include`.

If use is made of the `freadval` function in the IO library, the module initialiser function is extended (see Section 5.4 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 A). For example, suppose that module `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.

References

[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   The libCG.a Library

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

A.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-SL 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 1.16 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

The definitions in `cg_aux.h` contain auxiliary definitions which are dependent of the library used to implement the VDM-SL data types[5].

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

The class `vdmBase` is only used by code generated by the VDM++ version of the code generator.

```
/***
*  * 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
```

---

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

```
*  *    (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   Handcoded C++ Files

B.1   DefaultMod_userdef.h

```
#define TAG_DefaultMod 100
```

B.2   DefaultMod_userimpl.cc

We have chosen to implement `vdm_DefaultMod_ImplSort` as a handwritten version of `MergeSort`.

```
/***
*  * 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 Makefiles

C.1 Makefile for Unix Platform

```
#
# 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   Makefile for Windows Platform

```
#
# 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"

LDFLAGS = /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

CGOBJS = DefaultMod.obj

CGFLAGS = /D CG

sort_ex.exe: sort_ex.obj $(CGOBJS)

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 $@
```

48

```
################################################################
#### 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
```