VDMTools®

The VDM++ to Java Code
Generator

THE
IFAD®
VDMTools

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

Contents

1   Introduction

The VDM++ to Java Code Generator supports automatic generation of Java code from VDM++ specifications. The Code Generator provides a rapid way of implementing Java applications based on VDM++ specifications.

The Code Generator is an add-on feature to the VDM++ Toolbox. This manual is an extension to the *User Manual for the IFAD VDM++ Toolbox* [UserManPP] and gives an introduction to the VDM++ to Java Code Generator.

This manual is structured in the following way:

Section 2 gives an introduction to the VDM++ to Java Code Generator. It describes how to invoke the Code Generator from the VDM++ Toolbox and provides guidance on interfacing the generated Java code. Furthermore, it will be explained how to compile and run the Java code.

Section 3 presents four more advanced issues. It summarizes the options which can be chosen when generating Java code from VDM++ specifications. Moreover, it describes how to handle implicit or preliminary function/operation definitions and it discusses the possibilities for substituting generated Java code with handwritten code. Finally, it will list the requirements which a VDM++ specification must fulfil in order to be translated to compilable and correct Java code.

Section 4 gives a detailed description of the structure of the generated Java code. In addition, it explains the relation between VDM++ and Java data types, and it describes some of the design decisions made, when developing the VDM++ to Java Code Generator, including the name conventions used. This section should be studied intensively before using the Code Generator professionally.

Finally, in Section 5 an explanation of how to generate code for concurrent specifications is provided. For such specifications, multithreaded Java code is generated. As well as instructions on use, an overview of the translation approach is given.

2   The Code Generator - Getting Started

To get started using the Code Generator a VDM++ specification should be written in one or several files.

In the following, the VDM++ specification of a class `DoSort` will be used in order to illustrate the Java Code Generator. The specification is listed in Appendix D.1 and it can be found in file `Sort.rtf` provided in the distribution. In Section 2.1 it is explained how to generate Java code for the VDM++ `DoSort` class using the VDM++ Toolbox. In Section 2.2 it is explained how to write an application on top of the generated Java code. In Section 2.3 it is shown how to compile and run the application.

It is recommended that readers go through the steps described in Section 2.1 to 2.3 on their own computer.

2.1   Generating Code Using the VDM++ Toolbox

We will now describe how to use the VDM++ to Java Code Generator from the graphical user interface of the VDM++ Toolbox.

Having started the VDM++ Toolbox, a new project should be created the `Sort.rtf` file. Before generating Java code, it has to be ensured, that the VDM++ specification satisfies the necessary requirements:

- *All* files of the VDM++ specification in a project must have successfully been syntax checked in order to generate correct code for any selected class.

- 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.

Syntax and type check the `DoSort` class as described in the *User Manual for the IFAD VDM++ Toolbox* [UserManPP]. The result is shown in Figure 1.

---

[1]There exist two classes of well-formedness as explained in [LangManPP]. In the current context we mean possible well-formed type correctness.
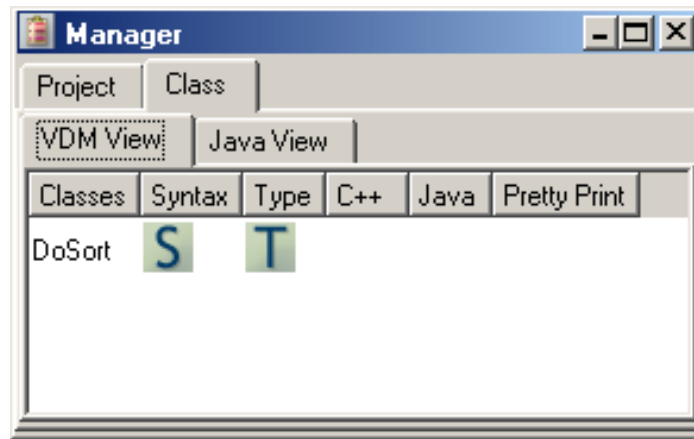
Figure 1: The Manager after Syntax and Type Checking

You can now generate code for the `DoSort` class by clicking on the ⬛ (Generate Java) button. In general, more than one file/class can be selected, in which case all of them are translated to Java.

Figure 2 shows how to generate Java code for the `DoSort` class. As can be seen, a Java file called `DoSort.java` has been code generated. It contains the Java class definition of `DoSort`. The `DoSort.java` file will be written in the directory, where the project file lies. If no project file exists, the file will be written in the directory, where the VDM++ Toolbox was started.

Figure 3 shows a skeleton of the VDM++ specification and the corresponding generated Java code for class `DoSort`. The different parts of the generated code will be explained in the following sections. The file `DoSort.java` is shown in full in Appendix D.2.
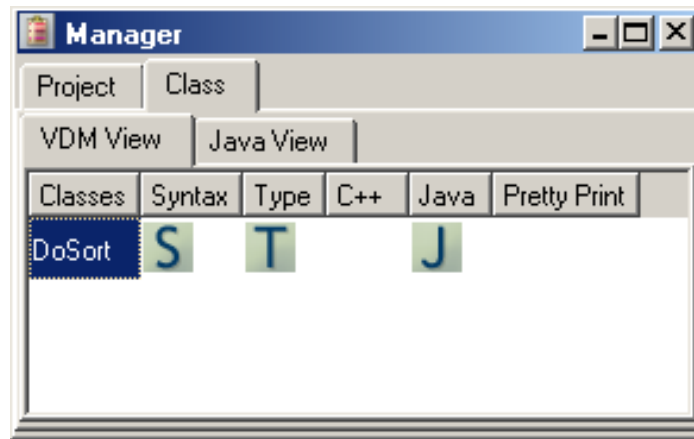
Figure 2: Code generating the `DoSort` class.

It is also possible to generate Java code from the command-line version of the VDM++ Toolbox. The VDM++ Toolbox is started from the command line with the command `vppde`. The `-j` option is used in order to generate Java code. To code generate the class `DoSort`, the following command is executed:

```
vppde -j Sort.rtf
```

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 Java files. For the described specification, the file `DoSort.java` containing the `DoSort` class definition will be generated.

Note: If a specification contains several classes and the command-line version of The Code Generator is used, all classes have to be code generated at the same time.

2.2   Interfacing the Generated Code

We have now reached the point, where Java code has been generated from a VDM++ specification. We will now show how to write an interface to the generated `DoSort` class in order to compile and run an application.

```
class DoSort

operations
  public Sort: seq of int ==> seq of int
  Sort(l) ==
    ...

functions

  protected DoSorting: seq of int -> seq of int
  DoSorting(l) ==
    ...

  private InsertSorted: int * seq of int -> seq of int
  InsertSorted(i,l) ==
    ...

end DoSort
```

VDM++

```
public class DoSort {

// ***** VDMTOOLS START Name=vdmComp KEEP=NO
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp


// ***** VDMTOOLS START Name=Sort KEEP=NO
  public Vector Sort (final Vector l) throws CGException{
    ...
  }
// ***** VDMTOOLS END Name=Sort


// ***** VDMTOOLS START Name=DoSorting KEEP=NO
  protected Vector DoSorting (final Vector l) throws CGException{
    ...
  }
// ***** VDMTOOLS END Name=DoSorting


// ***** VDMTOOLS START Name=InsertSorted KEEP=NO
  private Vector InsertSorted (final Long i, final Vector l)
                              throws CGException {
    ...
  }
// ***** VDMTOOLS END Name=InsertSorted
}
```

Figure 3: The VDM++ and the generated Java `DoSort` class.

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

```
01   Main() ==
02     let arr = [23,1,42,31] in
03     ( dcl res : seq of int = [],
04           dos : DoSort := new DoSort();
05       res = dos.Sort(arr);
06     )
```

We will now implement a Java main program with the same functionality as the above VDM++ specification. The Java file, containing the main program, should start by importing all classes of the VDM Java Library package `dk.ifad.toolbox.VDM`:

```
import dk.ifad.toolbox.VDM.*;
```

This saves the need to type fully qualified names for these classes. The VDM Java Library is described in more detail in Section 4.1. Let us now, step by step, translate the above listed VDM specification to Java.

Line 02 specifies an integer list. Translated to Java, one will get the following code:

```
Vector arr = new Vector();
arr.add(new Integer(23));
arr.add(new Integer(1));
arr.add(new Integer(42));
arr.add(new Integer(31));
```

The `Vector` class can be found in the `java.util` package. The `Sort` method of class `DoSort` expects an object of type `Vector` as input.

Line 03 declares a variable `res` of type `seq of int`, which will later be used to contain the sorted integer sequences. The Java code for this is just:

```
Vector res = new Vector();
```

Let us now show how to call the `Sort` method in class `DoSort`. Line 04 declares an object reference `dos` to an instance of the class `DoSort`, and line 05 calls the `Sort` method of the `DoSort` class with the integer sequence `arr` as argument. The result is assigned to `res`. Translated to Java, one will get the following code:

```
System.out.println("Evaluating Sort("+UTIL.toString(arr)+"):");
```

```
DoSort dos = new DoSort();
res = dos.Sort(arr);
System.out.println(UTIL.toString(res));
```

The `UTIL.toString` method, which is part of the VDM Java Library can be used in order to get a string containing an ASCII representation of a VDM value. This method is being used here to print relevant log messages to standard output during execution.

The above listed Java code has to be written in a `try` block in order to handle exceptions thrown by methods in the generated Java code. The `try` block is followed by a `catch clause`, that catches and handles these exceptions. All exceptions thrown by the generated Java code are subclasses of the `CGException` class, which again is part of the VDM Java Library. Thus the following `catch` statement is possible:

```
try {
...
}
catch (CGException e){
      System.out.println(e.getMessage());
}
```

The main program described above is implemented in the file named `MainSort.java` and it is listed in full in Appendix D.3.

2.3 Compiling and Running the Java Code

Having handwritten the main program, it is possible to compile and run the Java code.

Java code generated by this version of the VDM++ to Java Code Generator is compatible with the Java Development Kit version **1.3**.

The main program can be compiled by:

```
javac MainSort.java
```

Ensure that your `CLASSPATH` environment variable includes the VDM Java Library, i.e., the `VDM.jar` file. If you are using the Unix Bourne shell or a compatible shell, you can do this with the following commands:

```
CLASSPATH=VDM_Java_Library/VDM.jar:$CLASSPATH
export CLASSPATH
```

Replace `VDM_Java_Library` with the name of the directory in which the VDM Java Library is installed.

If you are working on a Windows-based system the `CLASSPATH` environment variable can be updated in `autoexec.bat` or from the `System` icon in the Control Panel. Note that for Windows you must use ";" and not ":" as the delimiter.

The main program `MainSort` can now be executed. Its output is listed below.

```
$  java MainSort
Evaluating Sort([23, 1, 42, 31]):
[1, 23, 31, 42]
$
```

In this section we have presented a brief introduction in how to use the Code Generator. In the following sections we describe different aspects of the Code Generator in more detail. Note that from now on, whenever portions of generated code are shown, only those parts relevant to the topic under discussion will appear in the text.

3   The Code Generator - Advanced Issues

Section 2 has given a short introduction to the Code Generator. This section will give the answer to the following questions:

- Which options can be chosen when generating Java code from VDM++ specifi-cations? (Section 3.1)

- What can be done if the specification contains implicit or preliminary func-tions/operations? (Section 3.2)

- What are the possibilities for substituting generated Java code with handwritten code? (Section 3.4)

- What requirements must a VDM++ specification fulfil to be translated to com-pilable and correct Java code? (Section 3.6)

3.1   Options of the VDM++ to Java Code Generator

When you generate Java code from your VDM++ specification you can choose one or more of the following options in order to influence the generated code. To view the options available, select the *Java Code Generator* entry from the options menu, as shown in Figure 4.
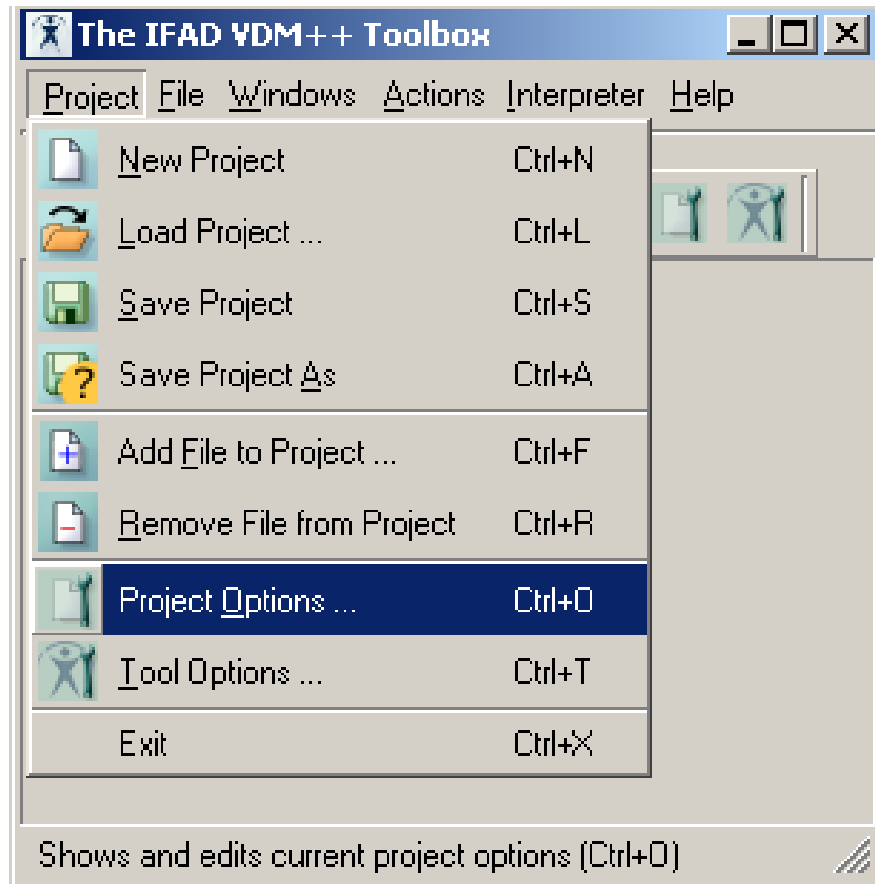
Figure 4: Selecting Java Code Generator Options

The various options available in the Code Generator are shown in Figure 5. Each of these options is described below. Note that all of these options are also available in the command-line version of the Code Generator. The appropriate flags are shown in brackets after the name of each option below. Default behaviour is also described with "off" meaning that by default the behaviour specified by the given option is not used, and "on" meaning such behaviour is used.

Code generate only skeletons, except for types (-s) Specify this option to generate skeleton classes. A skeleton class is a class containing full type, value and instance variable definitions, but empty function and operation definitions. Default: **off**

Code generate only types (-u) Specify this option to only want to generate Java code for VDM++ type definitions (i.e. functions, operations, instance variables
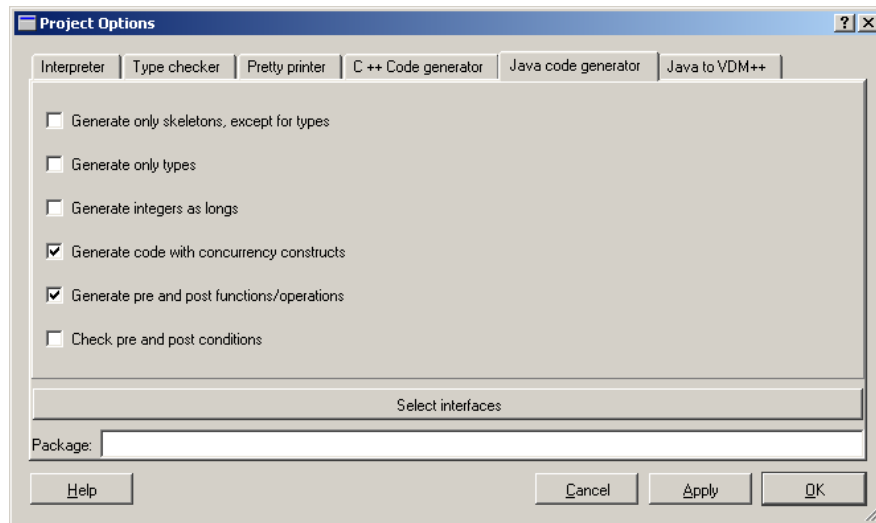
Figure 5: Options for Java Code Generation

and values will not be generated). Default: **off**.

Code generate integers as Longs (-L) Using this option, it is possible to generate VDM++ integer values and variables as Java `Long`s instead of `Integer`s. Default: **off**.

Code generate code with concurrency constructs (-e) This option is used to force the Code Generator to generate code which includes support for concurrency. See Section 5 for details of this. Default: **on**.

Code generate pre and post functions/operations (-k) Specify this option in order to code generate Java methods for pre and post conditions and invariants them. Default: **on**

Check pre and post conditions (-P) Specify this option to generate inline checks of function pre and post conditions, and operation pre conditions. Raise an exception if a check fails. This implies the previous option as pre and post conditions must be generated for compilable code to be generated. Default: **off**.

Package (-z) *packagename* The default behaviour of the code generator is to write the generated Java files in the directory, where your project file lies, or if no project file exists, in the directory, where the VDM++ Toolbox was started. The files are part of an unnamed default package. Specify this option in order to generate a specific package which will contain the generated Java classes. The Code Generator will make a new directory using the given package name containing the created files and the generated files will include the appropriate `package` statement.

Select Interfaces (-U) Select classes to be generated as Java interfaces. See Section 3.5 for details.

When starting the VDM++ Toolbox from the command line the following command has to be used:

```
vppde -j [options] specfile(s)
```

3.2 Implementing Implicit and Preliminary Functions/Operations

Implicit functions/operations and preliminary functions/operations (specified by "is not yet specified") are handled in the same way by the Code Generator. Look at the following VDM++ class definition containing a preliminary operation definition.

```
class A
operations
op:() ==> int
op() == is not yet specified;
end A
```

This class will be generated as follows:

```
public class A {
  protected external_A child = new external_A(this);
  private Integer op () throws CGException{
    return child.impl_op();
  }
};
```

As can be seen from the code listed above, the class `AA` contains a protected instance variable `child` of type `external_A`. This is the case for all classes containing implicit functions/operations or preliminary functions/operations (specified by "is not yet specified"). Though a class may have several of these definitions, there will only exist one instance of this external class.

The method `op` will call a method called `impl_op` on this instance.[2] The result of the `impl_op` method is returned as the result of the `op` method.

---

[2]`impl` stands for "to be implemented in Java".

It is then the user's responsibility to implement the method `impl_op` in class `external_A`. The input and output parameters of the method `impl_op` must be the same as those of the method `op`.

If a VDM++ class contains more than one implicit function/operation or preliminary function/operation (specified by "`is not yet specified`"), all methods have to be implemented in class `external_<CLASSNAME>`.

In order to make it easy for the user to implement the external class file, the Code Generator generates a file `external_A.java` for it. With the help of this file, the generated Java code will be compilable. However, a run-time error will occur, when a preliminary function is called. The file `external_A.java` containing the `external_A` class is listed below.

```
public class external_A {
  A parent = null;
  public external_A (A parentA) {
    parent = parentA;
  }
  public Integer impl_op () throws CGException{
    UTIL.RunTime("Preliminary Operation op has been called");
    return new Integer(0);
  }
};
```

The easiest way to implement the `external_A` class is to modify the template class, i.e. the user just has to replace the code

```
UTIL.RunTime("Preliminary Operation op has been called");
return new Integer(0);
```

with user-defined code, in the usual way in which user-defined code can replace generated code. (See Section 3.4 for details.)

Note, that the generated constructor for the external class takes an instance of the class `A` as input parameter and assigns it to the variable `parent`. In this way, the implementation of preliminary operation definitions can access the public state of the class `A`. Java methods for preliminary functions also use this constructor, though they are not allowed to act on the internal state of a class. They can however call an operation and thereby act on the internal state indirectly.

Implicitly defined functions and operations are handled in the same way as preliminary function and operation specifications containing the clause "`is not yet specified`".

Note, that the external class can contain implicit and preliminary operation and function definitions. In the generated template, they can be distinguished by the generated runtime error message:

```
UTIL.RunTime("Preliminary Operation op has been called");
```

for a preliminary operation definition called `op` and

```
UTIL.RunTime("Implicit Function f has been called");
```

for an implicit function definition called `f`, for example.

3.3   Generation of Abstract Classes

A VDM++ class is abstract if it contains preliminary function or operation definitions, or if it is a subclass of an abstract class, and does not provide implementations for the abstract functions and operations that have been inherited. Thus being abstract is an indirect property of a VDM++ class.

In contrast, Java provides a primitive notion of abstract classes. Thus when generating Java code, those VDM++ classes that are identified as being abstract, will be generated as abstract Java classes. For example, consider the VDM++ classes `A`, `B` and `C` below:

```
class A

instance variables
  protected m : nat := 1

operations
  public op : nat ==> nat
  op(n) == is subclass responsibility;

functions
  public f : int -> int
  f(i) == is subclass responsibility

end A

class B is subclass of A

operations
```

```
      public op : nat ==> nat
      op(n) ==
        return m + n


  end B


  class C is subclass of B


  functions
    public f : int -> int
    f(i) == i + 1


  end C
```

Class `A` contains preliminary functions and operations and is therefore abstract. It would therefore be code generated as:

```
public abstract class A {

  protected Integer m = null;
  public abstract Integer op (final Integer n) throws CGException;
  public abstract Integer f (final Integer i) throws CGException;

}
```

Class `B` inherits from abstract class `A`, and does not provide an implementation of the function `f`. Therefore it is also abstract:

```
public abstract class B extends A {

  public Integer op (final Integer n) throws CGException {
    return new Integer(m.intValue() + n.intValue());
  }
}
```

Finally, since class `C` inherits from `B` and provides an implementation of `f`. It is therefore a normal class:

```
public class C extends B {

  public Integer f (final Integer i) throws CGException{
    return new Integer(i.intValue() + new Integer(1).intValue());
```

```
        }
    }
```

3.4 Substituting Parts of the Generated Java Code

In a typical application, it will be necessary for the generated code to interact with other code e.g. external libraries and/or handwritten code. To facilitate such interaction, it is possible to modify the generated code, in such a way that these modifications are not overwritten if the Code Generator is rerun.

The way this is achieved is through the use of *keep tags*. These are comments in the generated Java code, which the Code Generator uses to decide whether a portion of the code should be overwritten or not.

For example, consider the following example:

```
class Date

types
  public Day = <Mon> | <Tue> | <Wed> | <Thu> | <Fri> | <Sat> | <Sun>;
  public Month = <Jan> | <Feb> | <Mar> | <Apr> | <May> | <Jun>
          | <Jul> | <Aug> | <Sep> | <Oct> | <Nov> | <Dec>;
  public Year = nat

instance variables
  d : Day;
  m : Month;
  y : Year

operations

  public SetDate : Day * Month * Year ==> ()
  SetDate(nd,nm,ny) ==
  ( d := nd;
    m := nm;
    y := ny );

  public today : () ==> Date
  today() ==
    return new Date()
end Date
```

Since neither VDM++ nor VDM++ Toolbox has a primitive notion of time, it is not

possible to give a complete specification of `today`. In the generated code, `today` is generated as follows:

```
// ***** VDMTOOLS START Name=today KEEP=NO
  public Date today () throws CGException{
    return (Date) new Date();
  }
// ***** VDMTOOLS END Name=today
```

The comments above and below the function definition represent the keep tag for this function. In a keep tag the following information is found:

- The name of the entity to which the tag applies (what constitutes an entity is explained below). This appears immediately after the text `Name=`.

- A flag indicating whether this entity should be retained or overwritten. This is given by the text after `KEEP=`. If it is `NO`, the entity will be overwritten; if `YES` it is retained. The default when a file is generated is `NO`.

Suppose we wish to modify this function to actually return the current day. This is possible using the `Calendar` class provided as part of the Java Development Kit.

```
// ***** VDMTOOLS START Name=today KEEP=YES
  public Date today () throws CGException{
    Calendar c = Calendar.getInstance();
    Date result = new Date();
    Object td = new Object(), tm = new Object();
    switch (c.get(Calendar.DAY_OF_WEEK)){
    case Calendar.MONDAY:
        td = new quotes.Mon();
        break;
    ...
    }
    switch (c.get(Calendar.MONTH)){
    case Calendar.JANUARY:
        tm = new quotes.Jan();
        break;
    ...
    }
    result.SetDate(td, tm, new Integer(c.get(Calendar.YEAR)));
    return result;
  }
// ***** VDMTOOLS END Name=today
```

First note that the keep tag has been changed to YES. This ensures that the changes made are preserved. The body of the function is then normal Java code, which is able to use arbitrary external classes.

In addition to changing existing entities, new entities can be added to the Java file. Suppose we wish to replace the default `toString` method (inherited from `java.lang.Object`) with one tailored to dates. We could add the following to the class definition.

```
// ***** VDMTOOLS START Name=toString KEEP=YES
  public String toString(){
      return d.toString() + m.toString() + y.toString();
  }
// ***** VDMTOOLS END Name=toString
```

3.4.1   Entities

An entity is a region in a generated Java file which can be retained using keep tags. It may be one of the following:

- A top-level class member variable.

- A top-level class method (including constructors).

- An inner class.

- A collection of import declarations.

- A package declaration.

- A header comment i.e. a region at the head of the file, in which comments can be placed, for instance version control information.

Note that keep tags may also be used with classes generated as interfaces (see Section 3.5); in that case the same rules apply, where interface should be read instead of class.

Three tag names are predefined and always appear in the generated file: `HeaderComment` for header comments, `package` for package declarations and `imports` for import declarations.

3.4.2   Rules for keep tags

The following rules must be followed when using keep tags.

- Each tag name must be unique.

- Keep tags must be flat i.e tags can not be nested.

- Outside a class definition, the only tags that may appear are `HeaderComment`, `package` and `imports`.

- Added entities must appear **within** the class definition, but at the top-level. Thus for instance if a function is added to an inner class, the whole inner class must be tagged `YES`.

- The syntax of keep tags is case and white-space sensitive. It must be followed exactly.

Failure to follow these rules could lead to code being overwritten. However since the original file is always backed up, this need not be fatal.

3.5   Generating Interfaces

The Code Generator allows generation of Java interfaces [Gosling&00]. A VDM++ class may be generated as an interface if the following conditions apply:

- All of the functions and operations defined in the class have body `is subclass responsibility`.

- The class contains no instance variables are defined in the class.

- All types defined in the class are public.

- All values defined in the class can be defined directly (see Section 4.5 for an explanation of what is meant by directly defined values).

- All superclasses of this class can be generated as interfaces.

For instance, consider the example in Figure 6. The class `A` clearly fulfils the requirements for being generated as an interface, since it has a directly defined value and all of its functions and operations are `is subclass responsibility`. Class `B` can also be generated as an interface, since it provides just one abstract function, and inherits from a class that can be generated as an interface. Class `C` however can not be generated as an interface, since it declares a non-abstract function.

To select which classes are to be generated as interfaces, click on the *Select Interfaces* button in the options dialogue box (as described in Section 3.1). A new dialogue box opens, as shown in Figure 7.

```
class A

values
  public v : nat = 1

operations
  public op : nat ==> nat
  op(n) == is subclass responsibility

functions
  public f : nat -> nat
  f(n) == is subclass responsibility

end A

class B is subclass of A

functions
  public g : nat -> nat
  g(n) == is subclass responsibility

end B

class C is subclass of A

functions
  public g : nat -> nat
  g(n) == n + 1

end C
```

Figure 6: Interfaces Example
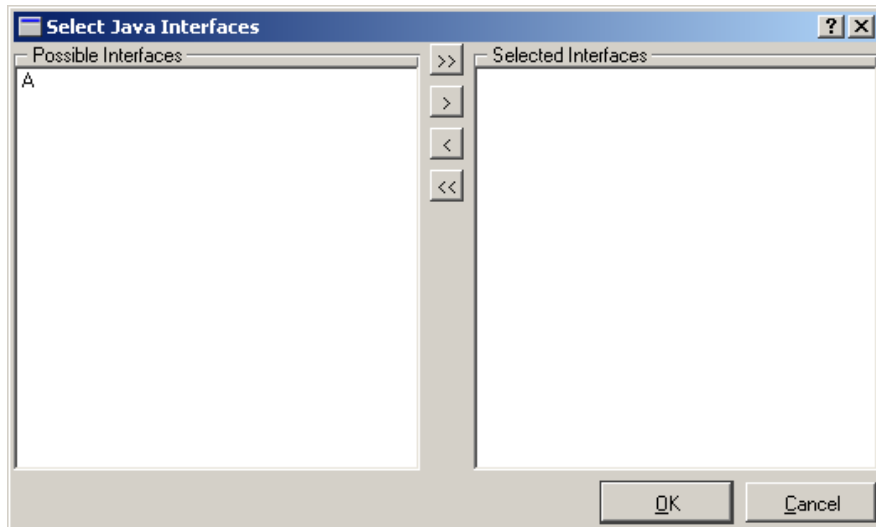
Figure 7: Initial Interface Selection Dialogue
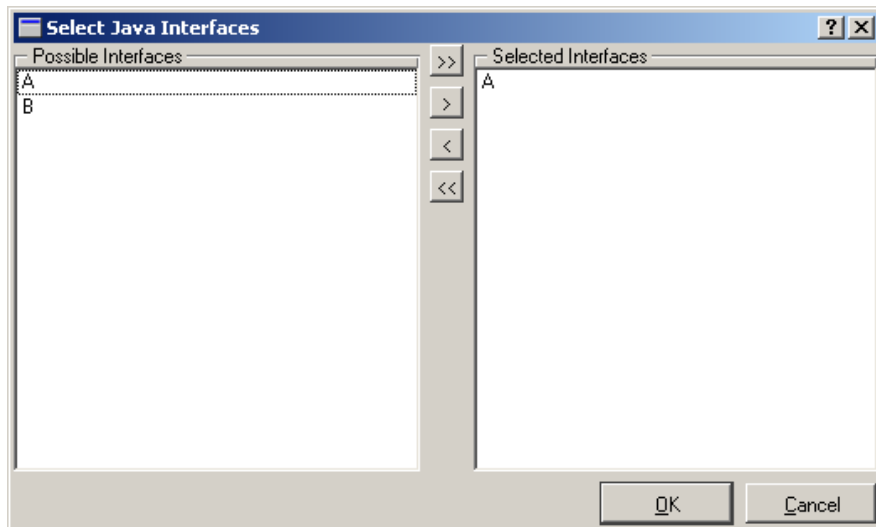
Figure 8: Updated Interface Selection Dialogue

Initially, only one class may be generated as an interface - `A`. If this is selected (by clicking on the *Add* button), the dialogue is updated as shown in Figure 8.

Note that since `B` now appears in the list of possible interfaces. This is because it can only be generated as an interface if its superclass - `A` - is an interface. If `A` is now removed from the list of select interfaces, `B` will automatically be removed from the list, since it no longer satisfies the criteria to be an interface.

Having selected classes to be generated as interfaces, code generation proceeds as normal. The following code would be generated for `A`:

```
public interface A {

// ***** VDMTOOLS START Name=v KEEP=NO
  private static final Integer v = new Integer(1);
// ***** VDMTOOLS END Name=v


// ***** VDMTOOLS START Name=op KEEP=NO
  public abstract Integer op (final Integer n) throws CGException;
// ***** VDMTOOLS END Name=op


// ***** VDMTOOLS START Name=f KEEP=NO
  public abstract Integer f (final Integer n) throws CGException;
// ***** VDMTOOLS END Name=f


}
```

Interfaces may also be selected using the command-line version of the Toolbox, using the `-U` option:

```
vppde -j -U class{,class} specfiles
```

If a class, which does not satisfy the above interface critera, is selected as an interface, the following error message will be generated:

```
Can not generate class class as an interface - ignored
```

3.6 Limitations

Not all VDM++ specifications can be code generated to Java. The VDM++ specifications have to meet certain requirements in order to be translated to compilable and correct Java code. These limitations are caused mainly by two reasons:

- Limitation of the translation algorithm used: VDM++ and Java are two different languages. In a small number of cases the translation of some VDM++ constructs can lead to incorrect Java code. The limitations caused by this fact are listed in Section 3.6.1. VDM++ specifications, that does not fulfil the listed requirements, can result in non-compilable and incorrect Java code. The Code Generator generates a warning/error message when it encounters a VDM++ feature not translatable to Java.

- Limitation of the domain of the translation: The Code Generator does not support all VDM++ constructs. Section 3.6.2 summarizes the VDM++ constructs not supported by the Code Generator. These constructs do not result in uncompilable Java code, but the execution of the generated code for these constructs will result in run-time errors. The Code Generator will give a warning whenever an unsupported construct is encountered.

Note that the semantics of Java and VDM++ differ with respect to how private methods are handled with respect to dynamic dispatch. Consider the following example:

```
class C                          class D is subclass of C
operations                       operations
public op1 : () ==> seq of char  public op3 : () ==> seq of char
op1() == op2();                  op3() == op1();

private op2 : () ==> seq of char private op2 : () ==> seq of char
op2() == return "C'op2"          op2() == return "D'op2"
end C                            end D
```

In Java, the expression `new D().op3()` yields the result `C'op2`. In VDM++ the same expression yields `"D'op2"`.

3.6.1 Requirements of VDM++ specifications due to language differences

The VDM++ specification has to meet the following requirements in order to generate *compilable* and *correct* Java code:

- Type checker warnings as "*Missing type information*" should be removed, because they can lead to errors in the generated code. The Code Generator is not able to generate correct Java types, if type information is missing for a VDM construct.

- Classes, instance variables, types, values, functions and operations may not have the same name. Moreover, redeclaration of names should be avoided. That means, the following VDM++ specification for example will result in non-compilable code because the variable name `a` is redeclared:

    ```
    f : int | (int * int) ==> bool
    ```

```
f(a) ==
  cases a:
    2 -> return true,
    mk_(a,b) ->  return false,
    others -> let a = 1 in return true
  end;
```

- Abstract operations/functions must have the same type as the operations/functions implementing them. Consider the following example:

```
class A
operations
  m: nat ==> nat
  m(n) == is not yet specified;
end A

class B is subclass of A
operations
  m: nat ==> nat
  m(n) = return n+n;
end B
```

If the type of B'm did not exactly match that of A'm, then A'm would still be abstract in B, and therefore B would be an abstract class.

- A limited form for multiple inheritance may be used. However, the classes involved have to fulfil the conditions described in Section 3.5.

- If all branches in a case statement contain a return statement, the case statement must have an others branch. Otherwise the Java compiler generates a "*Return required*" error when compiling the generated Java code.

- Dead code should be avoided. Consider the following example:

```
operations
  m : nat ==> nat
  m(n) ==
    (return n;
     a:= 4;
    );
```

The statement a:= 4; will never be executed, which leads to an "*Statement not reached*" error when compiling the generated Java code.

- When operation calls in a superclass are qualified by name, the generated code can be erroneous. Look at the following example:

```
class A

operations
```

```
      public SetVal : nat ==> ()
      SetVal(n) == ...;

      end A

      class B is subclass of A

      operations

      public SetVal : nat ==> ()
      SetVal(n) == ...

      end B

      class C is subclass of B

      operations

      public Test : () ==> ()
      Test() ==
        ( self.SetVal(1);
          self.B'SetVal(1);
          self.A'SetVal(2)
        )

      end C

      class D

      instance variables
        b : B := new B()

      operations

      public Test: () ==> ()
      Test() ==
        (b.SetVal(1);
         b.B'SetVal(5);
         b.A'SetVal(2)
        )

      end D
```

Let us start looking at class `C`: The statement `self.SetVal(1)` calls the `SetVal` operation in class `C` and will be code generated as `this.SetVal(1)` in Java. The statement `self.B'SetVal(1)` calls the `SetVal` operation in class `B` and will be code generated as `super.SetVal(1)` in Java. In Java it is impossible to call the `SetVal()` method in class `A`. The statement `self.A'SetVal(2)` will be code

generated as `super.SetVal(2)`. If there was no `SetVal` operation in class B, this would be correct. However, in the above case, this is not in conformity with the VDM++ specification. The two operation calls `self.B'SetVal(1)` and `self.A'SetVal(2)` will cause the Code Generator to give the warning "*Quoted method call is code generated as a call to super*". The user can then make sure if the correct method is called.

Let us now look at class D: The statement `b.SetVal(1)` calls the `SetVal` operation in class B and will be code generated as `b.SetVal(1)` in Java. In Java it is not possible to invoke overriden methods from outside the class that does the overriding. There is therefore no way to call the `SetVal` method in class A. The quoted operation calls in class D are therefore all code generated as `b.SetVal(1)`. The code generating will however give the warning "*Quoted method call is removed*" in order to inform the user.

- The max (min) values for integer and double types in Java are smaller (bigger) than the respective values in VDM++. Values, that are not valid in Java lead to errors when running the generated Java code.

3.6.2 Unsupported Constructs

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

- Expressions:

  - Lambda.
  - Compose, iterate and equality for functions.
  - Type judgement expressions.
  - Higher order functions.
  - Local function definitions.
  - 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:

    - Specification statements.
    - 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.

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
```

The code generated for `f` will be compiled. The compiled Java code corresponding to `f` however 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. Generating code for the function `f` listed above leads to the *Error* window shown in Figure 9.
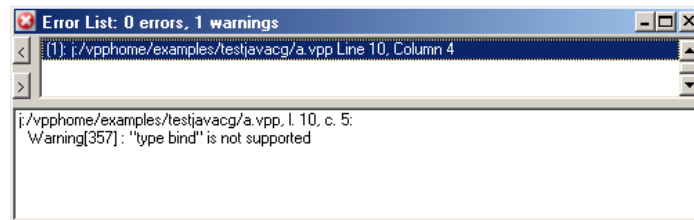
Figure 9: A warning generated by the Code Generator.

4 Code Generating VDM++ Specifications

This section will give you a detailed description of the way VDM++ constructs are code generated, including classes, types, values, instance variables, functions, operations, expressions and statements. This description should be studied intensively by those wishing to use the Code Generator professionally.

We will start by giving an introduction to the VDM Java Library, which forms the basis of code generating VDM++ specifications. Afterwards, we will describe the code generated for the above mentioned VDM++ constructs, one by one.

4.1 The VDM Java Library

The data refinement of the generated code is based on the VDM Java Library, which is implemented in the package `dk.ifad.toolbox.VDM`. Here, we will only give a short introduction to this library. It is further described by HTML documentation generated by the *javadoc* program. In order to get a full understanding of this library you should read that documentation. See Appendix B for a description about how to generate the HTML documentation using the *javadoc* program.

The VDM Java Library provides a fixed implementation of the following VDM++ data types:

- Product/Tuple Type
- Record Type

For each of these types a class has been implemented providing the same public methods as the corresponding VDM++ type. These classes are implemented on top of classes provided by the Java language.

VDM++ data types, which are not listed above (the basic VDM++ data types, sets, sequences, maps, the `Optional` type and the `ObjectReference` type) are represented by classes/constructs which are part of the Java language itself, or part of the standard Java Development Kit (JDK) distribution.

In addition to providing an implementation of the above listed VDM++ data types, the VDM Java Library provides two more classes:

- The `UTIL` class.

  This class contains auxiliary methods, which are used in the generated code and which can be used by the user when interfacing the generated code. The most important of these auxiliary methods are listed below:

– `clone`: clones (in-depth) a VDM value. However, VDM++ classes and basic VDM++ data types are not cloneable.
– `equals`: compares two VDM values.
– `toString`: returns a String containing an ASCII representation of a VDM value.
– `RunTime`: is called when a run-time error occurs. It throws a `VDMRunTimeException`, which is defined in the VDM Java Library.
– `NotSupported`: is called when an unsupported construct is executed. It throws a `NotSupportedConstructException`, which is defined in the VDM Java Library.

Note: Use always the `clone`, `toString` and `equals` methods of the `UTIL` class - and not the methods defined in the Java classes corresponding to VDM++ data types.

• The `CGException` class and its subclasses.

The error handling of the VDM Java library is based on Java's exception handling mechanism. When an error is detected by the generated Java code or one of the library methods, an appropriate exception is thrown. All the implemented errors are subclasses of the class `CGException`, which again is a subclass of the `java.lang.Exception` class. The inheritance structure of the exception classes is shown in Figure 10.

The different kinds of exceptions are grouped into two types.

– Instances of the `VDMRunTimeException` class: They are thrown in the generated Java code. They correspond to run-time errors occuring when executing VDM++ specifications.
– Instances of the `NotSupportedConstructException` class: They are thrown when constructs not supported by the Code Generator are executed.

4.2 Code Generating Classes

For each VDM++ class a corresponding Java class is generated. For each VDM++ class member, the corresponding item in the Java class will have the same access modifier as the VDM++ member.

Let us have a closer look at the structure of a Java class generated for a class in the VDM++ specification.

The generated Java class contains:

• A static *comparator*, implementing the interface `java.util.Comparator` in the Java Development Kit. This is used in tree-based data structures, and implements the VDM notion of equality.
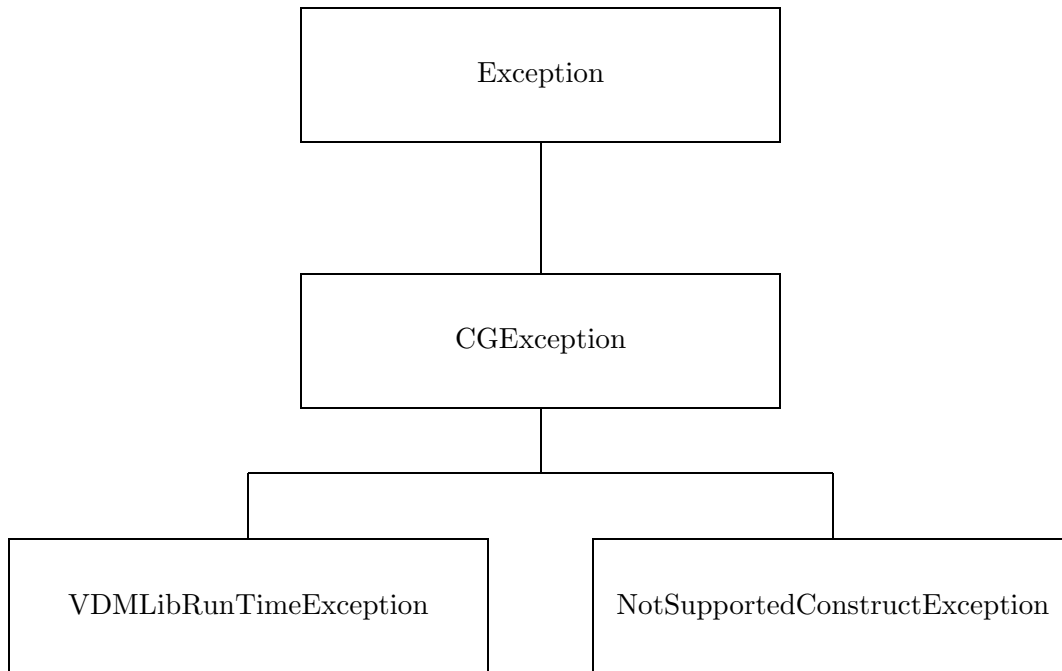
Figure 10: Inheritance structure of the Java classes handling Code Generator exceptions.

- Java code implementing VDM++ datatypes. (See Section 4.4)

- Java code implementing VDM++ values. (See Section 4.5)

- Java code implementing VDM++ instance variables. (See Section 4.6)

- A static initializer (if values have to be initialized).

- A constructor (if the class contains instance variable definitions).

- Java methods implementing VDM++ functions. (See Section 4.7)

- Java methods implementing VDM++ operations. (See Section 4.7)

- Code for concurrency (synchronization, threads etc), if that option is selected; see Section 5.

Consider the resulting skeleton of a generated Java class, generated for a VDM++ class definition, say `A`:

```
public class A {
```

```
// ***** VDMTOOLS START Name=vdmComp KEEP=NO
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp

  ...Implementation of VDM++ types...
  ...Implementation of VDM++ values...
  ...Implementation of VDM++ instance variables...

// ***** VDMTOOLS START Name=static KEEP=NO
  static {
         ...Initialization of VDM++ values...
         }
// ***** VDMTOOLS END Name=static

// ***** VDMTOOLS START Name=A KEEP=NO
  public A () {
      try { ...
            Initialization of VDM++ instance variables...
            ...
          }
      catch (Throwable e) { ...
                            }
  }
// ***** VDMTOOLS END Name=A

  ...Implementation of VDM++ functions...
  ...Implementation of VDM++ operations...

};
```

If a VDM++ class is abstract, the generated Java class will also be declared as such.

4.3   Inheritance Structure of the Generated Java Classes

The inheritance structure of the generated Java classes corresponds exactly to the inheritance structure of the VDM++ classes.

The inheritance structure of the VDM++ classes and the generated Java classes for the sorting example is shown in Figure 11.

Sorter

| Merge Sort | ExplSort | ImplSort | DoSort |

(a) VDM++

Object

Sorter

| Merge Sort | ExplSort | ImplSort | DoSort |
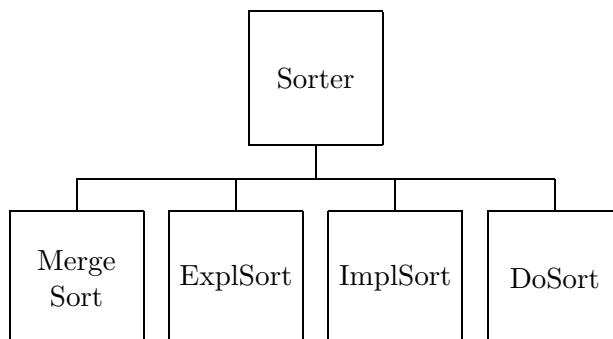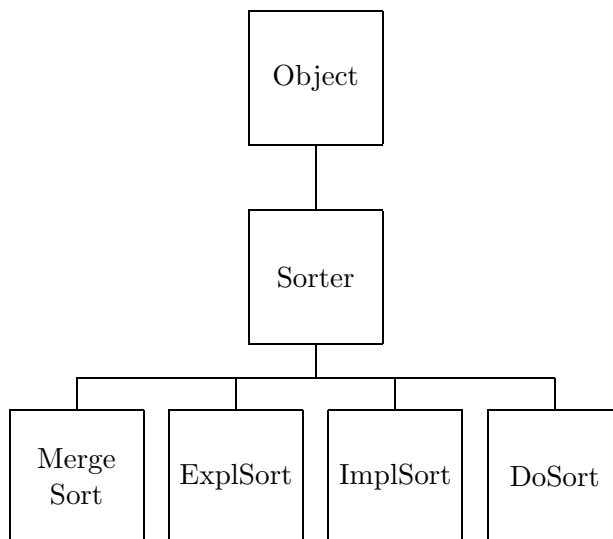
(b) Java

Figure 11: Inheritance structure of the VDM++ classes and the generated Java classes

VDM++ allows classes to have more than one superclass, using multiple inheritance. Java does not support multiple inheritance. Instead, Java replaces multiple inheritance with interfaces [Gosling&00]. A class in Java optionally `extends` one superclass and it optionally `implements` one or more interfaces. In order to implement an interface, a class must first declare the interface in an `implements` clause, and then it must provide an implementation for all the abstract methods of the interface. This is actually the real difference between multiple inheritance in VDM++ and interfaces in Java. In Java, a class can inherit actual implementations only from one superclass. It can inherit addi-

tional `abstract` methods from interfaces, but it must provide its own implementation of these methods.

To resolve multiple inheritance at the VDM++ level, the user must select which classes are to be code generated as interfaces (see Section 3.5 for details of how this is done). Since Java's interface model is more simple than the VDM++ multiple inheritance model, not all cases of multiple inheritance in VDM++ can be suitable resolved. In such circumstances the VDM++ model must be modified if complete code generation is desired.

In order to generate Java code for multiple inheritance in VDM++ the superclasses in VDM++ must fulfil the following conditions:

- Only one superclass may define functions and operation implementations, and only this superclass may provide instance variables. This class will be code generated as the single superclass in Java.

- It must be possible to generate all other superclasses as interfaces (see Section 3.5).

Note that if the subclass does not provide implementations for all abstract functions and operations that are inherited, it will be generated as an abstract class.

Consider the following example of a VDM++ specification, that can be code generated:

```
class E

instance variables
  protected i : nat

end E

class F

values
  public n : nat = 3

operations
  public  getx : () ==> nat
  getx() == is subclass responsibility;

end F

class G is subclass of E, F

operations
  public  getx : () ==> nat
  getx() == if true then return n else return i;
```

```
    end G
```

The listed VDM++ specification fulfils the necessary conditions: Class `G` is the subclass of the two classes `E` and `F`. Class `E` defines an instance variable. Therefore it will be code generated as the single superclass in Java. Class `F` may be generated as an interface, since it fulfils the criteria given in Section 3.5.

The generated Java code for `G` is listed below.

```
    public class G extends E implements F {

      static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();

      public Integer getx () throws CGException{
        if (new Boolean(true).booleanValue())
          return n;
        else
          return i;
      }
    };
```

If the VDM++ specification does not fulfil the above listed requirements, the VDM++ to Java Code Generator will generate incorrect code.

If multiple inheritance at the VDM++ level is not resolved, the Code Generator will result in the following error:

```
Error : "Multiple inheritance in this form" is not supported and is not
        code generated
```

Note: The VDM++ expressions *Base Class*, *Class*, *Same Base Class* and *Same Class Membership* will have a different semantics for generated Java code compared to the original VDM++ specification, in the presence of multiple inheritance.

4.4   Code Generating Types

In this section the way VDM++ types are mapped into Java code is described. In addition, the naming conventions for types are summarized.

4.4.1   Mapping Anonymous VDM++ Types to Java

Anonymous types are types that are not given a name in the VDM++ specification. The way in which they are code generated is described in the following sections.

The Boolean, Numeric and Character Types   The Java language package provides the following "wrapper" classes for the primitive data types double, int, boolean and char: `Double`, `Integer`, `Boolean` and `Character` respectively. These classes are used to represent the following VDM++ datatypes: `real`, `rat`, `int`, `nat`, `nat1`, `bool`, `char`. The VDM `real` and `rat` types are mapped to the Java class `Double`. The VDM `nat`, `nat1` and `int` types are mapped to the Java class `Integer`. The VDM `bool` type is mapped to the Java class `Boolean`. The VDM `char` type is mapped to the Java class `Character`.

Note that there is a semantic difference here between VDM++ and Java. In VDM++ the `int`, `nat`, `nat1` types are subtypes of the `real` and `rat` types. This means, that it is possible to assign an integer to a variable of type `real` and it is possible to assign a real to a variable of type `int`, if its value is an integer value.

In Java objects of type `Double` and `Integer` cannot be cast to each other in the same way. Therefore, two auxiliary Java methods: `UTIL.NumberToDouble` and `UTIL.NumberToInteger` have been provided. The `Number` class is a superclass to both the `Double` and the `Integer` class.

The Quote Type   The Code Generator generates a class definition for every quote used in the VDM++ specification. All quotes are collected in the `quotes` package. The quote `<HELLO>` will lead to the `HELLO` class definition in file `HELLO.java` in the package `quotes`:

```
package quotes;

public class HELLO {

  static private int hc = 0;

  public HELLO () {
    if (hc == 0)
      hc = super.hashCode();
  }

  public int hashCode () {
    return hc;
  }

  public boolean equals (Object obj) {
    return obj instanceof HELLO;
  }

  public String toString () {
    return "<HELLO>";
  }
```

```
    };
```

The quote `<HELLO>` can then be code generated as follows:

```
    new quotes.HELLO()
```

Note that the `hashCode` method ensures that every instance of a quote constant has the same hash code.

The Token Type   The Code Generator generates a `Token` class in the file `Token.java` when the VDM++ specification contains a token type:

```
    import dk.ifad.toolbox.VDM.*;

    public class Token {

      Object vdmValue;

      public Token (Object obj) {
        vdmValue = obj;
      }
      public Object GetValue () {
        return vdmValue;
      }
      public boolean equals (Object obj) {
        if (!(obj instanceof Token))
          return false;
        else
          return UTIL.equals(this.vdmValue, ((Token) obj).vdmValue);
      }
      public String toString () {
        return "mk_token(" + UTIL.toString(vdmValue) + ")";
      }
    };
```

The token value `mk_token(<HELLO>)` is for example code generated as follows:

```
    new Token(new quotes.HELLO());
```

The Sequence, Set and Map Types   The VDM Sequence (except the `seq of char` type), Set and Map types are mapped to the `Vector`, `TreeSet` and `HashMap` classes of the `java.util` package. For `TreeSet`s, comparison is based on the comparator defined

in the `UTIL` class provided. These classes respectively implement the interfaces `List`, `Set` and `Map`, also defined in the `java.util` package.

The `seq of char` type is mapped into the Java language class `String`. Note that for example the "`(seq of char | seq of nat)`" and the "`seq of (char | nat)`" types are generated as `Vector`.

The Tuple/Product Type The values of a product type are called tuples. The class, which models the VDM Tuple Type, is called `Tuple` and can be found in the VDM Java Library.

Note, that both the VDM++ types, i.e. `int * real` and `seq of nat * nat` are simply code generated as `Tuple`.

The Union Type Anonymous VDM++ Union types are supported by the Java `Object` class.

The Optional Type The VDM Optional Type is represented by the fact that object references may be "null" in Java.

The Object Reference Type In Section 4.2 it has been described, how a Java class is generated for each VDM++ class. In VDM++ an object reference type is denoted by a class name. The class name in the object reference type must be the name of a class defined in the specification. Moreover, in VDM++ a value of the object reference type can be regarded as a reference to an object.

The object reference type corresponds to Java's class/instance scheme. Java manipulates objects "by reference" as is the case for VDM++.

The Function Type The VDM++ Function Type is not supported in the Code Generator.

4.4.2   Mapping VDM++ Type Definitions to Java

For VDM++ record types, and union types consisting entirely of records, VDM++ to Java Code Generator generates inner classes representing the types. For other kinds of type definition, it is not necessary to generate a Java representation, since all other type definitions are shallow definitions. That is, they simply represent a new name for an existing type. In such cases VDM++ to Java Code Generator instead uses the

existing type, and the new name is not used. To illustrate this, consider the following example:

```
types
 A = nat
 B = seq of char
 C = A | B
```

The new types will always be equal to the types on the right hand side. Thus, they are just new names for the existing types on the right hand side. Therefore, the generated code will use the Java implementation of these right hand side types instead. When the types `A`, `B` or `C` are used in the VDM++ specification, they will be mapped to the Java classes `Integer`, `Vector` and `Object` respectively.

However, the Record types and Union types composed of Record types represent deep type definitions. That is, they introduce new types to the model. Therefore they are code generated in the manner described below:

- *The Composite/Record Type*

  All record types defined in a VDM++ specification are mapped to class definitions, that implement the `Record` interface found in the VDM Java Library. Fields in a record become variables in the new class.

  For example, the following composite type[3]

  ```
  public A::     real
              k : int
  ```

  will be code generated as:

  ```
  public static class A implements Record {

      public Double f1;
      public Integer k;

      public A () {}
      public A (Double p1, Integer p2){
        f1 = p1;
        k = p2;
      }
      public Object clone () {
        return new A(f1,k);
      }
      public String toString () {
        "mk_G'A(" + UTIL.toString(f1) + "," + UTIL.toString(k) + ")";
  ```

---

[3]Note: Do not use the `compose of` syntax to define composite types.

```
      }
      public boolean equals (Object obj) {
        if (!(obj instanceof A))
          return false;
        else {
          A temp = (A) obj;
          return UTIL.equals(f1, temp.f1) && UTIL.equals(k, temp.k);
        }
      }
      public int hashCode () {
        return (f1 == null ? 0 : f1.hashCode()) +
               (k == null ? 0 : k.hashCode());
      }
    };
```

For each field in the record, a public instance variable has been added to the generated class definition. The names of these variables 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. `f1` in the example above. If `f1` is already used as a field selector, then the character "f" will be repeatedly appended until a unique field selector is obtained.

- *Union Types composed of composite types*

  Union types, that are composed of composite types are code generated using Java interfaces. Look at the following VDM++ types:

  ```
  Item = MenuItem | RemoveItem;
  MenuItem = Seperator | Action;
  Action:: text: String;
  Separator::;
  RemoveItem::;
  ```

The generated Java code looks as:

```
    public static interface Item {
    };

    public static interface MenuItem extends Item {
    };

    private static class Action implements MenuItem , Record {
      ...
    } ;

    private static class Separator implements MenuItem , Record {
      ...
    } ;
```

```
private static class RemoveItem implements Item , Record {
  ...
} ;
```

As you can see, the classes generated for Record types implement the interfaces generated for the Union types.

4.4.3 Invariants

When an invariant is used to restrict a VDM++ type definition in the specification, an invariant VDM++ function is also available. This invariant function can be called in the same scope as its associated type definition (see [LangManPP]). When the option for generating pre and post functions/operations is chosen, the VDM++ to Java Code Generator generates a Java method definition corresponding to such an invariant function. As an example, consider the following VDM++ type definition:

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

The method declaration corresponding to the VDM++ function inv_S is listed below.

```
public Boolean inv_S(final TreeSet s) throws CGException {
...
};
```

Note, that the VDM++ to Java Code Generator does not support dynamic check of invariants, but invariant functions can be called explicitly.

4.5 Code Generating Values

VDM++ value definitions are translated to static final variables of the generated Java class. The static keyword is used to indicate that a particular variable is a class variable rather than an instance variable. Moreover, the final keyword indicates, that the variable is a constant.

Consider the example below:

```
class A
values
  public mk_(a,b) = mk_(3,6);
```

```
        private c : char = 'a';
        protected d        = a + 1;
        e        = 2 + 1;
    end A
```

The generated class variables in the Java class `A` will look like:

```
public class A {
    public static final Integer a;
    public static final Integer b;
    private static final Character c = new Character('a');
    protected static final Integer d;
    private static final Integer e = new Integer(new Integer(2).intValue() +
                                                 new Integer(1).intValue());
}
```

If the VDM++ values are initialized by a simple expression, that in addition does not contain any other VDM++ values, the corresponding Java variables are initialized "directly". As the example shows, the variables `c` and `e` are intialized directly. The other variables are initialized in the static initializer of the class. The static initializer is an initialization method for class variables. It is invoked automatically by the system when the class is loaded. The instance variables `a`, `b` and `d` are thus initialized in the static initializer of the generated Java class `A`. The static initializer for class A is listed below:

```
static {
    Integer atemp = null;
    Integer btemp = null;
    Integer dtemp = null;

    /** Initialization of class variables a & b */
    boolean succ_2 = true;
    {
      try{
        Tuple tmpVal_1 = new Tuple(2);
        tmpVal_1 = new Tuple(2);
        tmpVal_1.SetField(1, new Integer(3));
        tmpVal_1.SetField(2, new Integer(6));
        succ_2 = true;
        {
          Vector e_l_7 = new Vector();
          for (
          int i_8 = 1; i_8 <= tmpVal_1.Length(); i_8++)
            e_l_7.add(tmpVal_1.GetField(i_8));
          if (succ_2 = 2 == e_l_7.size()) {
```

```
             atemp = UTIL.NumberToInt(e_l_7.get(0));
             btemp = UTIL.NumberToInt(e_l_7.get(2 - 1));
          }
        }
        if (!succ_2)
          UTIL.RunTime("Pattern match did not succeed in value definition");
     }
     catch (Throwable e) {
       System.out.println(e.getMessage());
     }
   }
   a = atemp;
   b = btemp;

   /** Initialization of class variable d */
   {
     try{
       Integer tmpVal_11 = null;
       tmpVal_11 = new Integer(a.intValue() + new Integer(1).intValue());
       dtemp = tmpVal_11;
     }
     catch (Throwable e) {
       System.out.println(e.getMessage());
     }
   }
   d = dtemp;
 }
```

4.6 Code Generating Instance Variables

The code generation of instance variables is very straightforward. Instance variables
are translated into member variables of the corresponding Java class.

Consider the following instance variable declaration in VDM++:

```
class A
instance variables
  public i : nat;
  private k : int := 4;
  protected message : seq of char := [];
  inv len message <= 30;
  j : real := 1;
...
end A
```

The corresponding Java code generated by the Code Generator in file `A.java` will

become:

```
public class A {
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
  public Integer i = null;
  private Integer k = null;
  protected String message = null;
  private Double j = null;
  ...
}
```

Instance variables are initialized when an object is created. In Java, instance variables are initialized in the constructor methods, which are run when an instance of the class is created.

Thus, the implementation of the constructor method for class A initializes the instance variables as shown below:

```
public class A {
  public A () {
    try{
      k = new Integer(4);
      message = UTIL.ConvertToString(new String());
      j = UTIL.NumberToReal(new Double(1));
    }
    catch (Throwable e) {
      System.out.println(e.getMessage());
    }
  }
  ...
}
```

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

4.7 Code Generating Functions and Operations

In VDM++, functions and operations can be defined both explicitly or implicitly. The VDM++ to Java Code Generator generates Java methods for both implicit and explicit function and operation definitions.

In both VDM++ and Java all functions and operations are virtual, so there is no difference in semantics. The access modifier given to a generated method will be the

same as the corresponding VDM++ function or operation. A function or operation name in a VDM++ specification will be given the same name in the corresponding Java implementation.

All generated methods throw the `CGException` exception. This is done in order to handle exceptions thrown in the generated Java code.

4.7.1   Explicit Function and Operation Definitions

Let us look at an example for code generating explicit VDM++ function and operation definitions.

The operation definition `Sort` in the VDM++ class `DoSort` is explicit and leads to the following Java method in class `DoSort` in file `DoSort.java`:

```
public Vector Sort (final Vector l) throws CGException{
   ...
}
```

4.7.2   Preliminary Function and Operation Definitions

The body of explicit function and operation definitions can be specified in a preliminary manner using the clauses "`is subclass responsibility`" and "`is not yet specified`".

The "`is subclass responsibility`" clause indicates that implementation of this body must be undertaken by any subclasses. Preliminary function/operation specifications containing the clause "`is subclass responsibility`" are translated into abstract methods in Java. A Java class containing an abstract method is an abstract class. All derived classes will remain abstract until all abstract methods are implemented. In order to generate correct Java code, abstract operations/functions in the VDM++ specification must have the same input and output parameters as the operations/functions implementing them. Subclasses, which do not implement abstract methods will be generated as abstract classes.

The "`is not yet specified`" clause indicates that the implementation of this body must be undertaken by the user. In section 3.2 it has been described how this is done.

4.7.3   Implicit Function and Operation Definitions

The implementation of implicit functions and operation definitions has to be undertaken by the user. See Section 3.2 for more information.

4.7.4 Pre and Post Conditions

When pre and post conditions are specified for functions, corresponding pre and post methods can be generated by the VDM++ to Java Code Generator. Moreover, methods can be generated for pre conditions of operations. Post conditions on operation specifications, however, are ignored by the VDM++ to Java Code Generator. The "Code generate pre and post functions/operations" option has to be selected in order to generate Java method definitions corresponding to pre and post conditions.

The generated pre and post methods take the same access modifier as that of the corresponding function or operation. Their name is prefixed by `post` and `pre` respectively and their return type will always be `Boolean`. The "Check pre and post conditions" option can be used to generate code that checks pre and post conditions (not including operation post conditions).

4.8 Code Generating Expressions and Statements

VDM++ expressions and statements are code generated, so that the generated code behaves as intended by the specification.

The undefined expression and the error statement are translated into a call of the function `UTIL.RunTime`, found in the VDM Java Library, which throws a `VDMRunTimeException`.

4.9 Name Conventions

The naming strategy used by the VDM++ to Java Code Generator is to keep the same names as those being used in the VDM++ specification. This strategy applies to all identifiers used in the VDM++ specification. However, underscores ('_') and single quotes ('') appearing in identifiers will be exchanged with underscore-u ('_u') and underscore-q ('_q'), respectively, in the generated Java code. Moreover, reserved words, reserved method names, and names of classes in the `java.lang` package are prefixed by '`vdm_`'. Problems resulting from the redeclaration of variable names are solved by postfixing variable names with _number. Finally, auxiliary/temporary variable names are named as `name_number`.

5   Code Generation of Concurrent VDM++ Specifications

5.1   Introduction

VDM++ provides a number of features for specifying systems with concurrently executing threads. These allow specification of the functionality of individual threads, and specification of synchronization for objects shared amongst threads.

Java provides support for threads via the `Thread` class, and allows synchronization of shared objects using monitors. However VDM++ provides a more sophisticated mechanism for synchronizing access, so the translation from VDM++ specifications to Java is somewhat more subtle than might be expected.

5.2   Overview

In addition to the code generation described in the preceding chapters, the Concurrent VDM++ to Java Code Generator allows generation of the following constructs:

- procedural threads

- periodic threads

- the *start* statement

- permission predicates

- mutex synchronization

- history expressions

5.2.1   Code Generation

From the graphical user interface of the VDM++ Toolbox, the "Generate code with concurrency constructs" option should be selected. From the command line the `-e` flag should be used to specify generation of concurrency constructs:

```
vppde -j -e [other options] specfile(s)
```

5.3   Translation Approach

Code generation of concurrent VDM++ specifications is less straightforward than code generation of sequential specifications largely because mechanisms for synchronization

need to be implemented. In particular the translation approach needs to ensure that operation calls honour any synchronization constraints. This implies that the translation approach needs to provide a means of recording the information required to evaluate permission predicates, and in particular the history counters for a particular operation.

In the following we describe the core translation which takes place for each class. We then describe the extensions to this if a procedural or periodic thread is specified.

The knowledge in these sections is not needed to use the Concurrent VDM++ to Java Code Generator, so these sections may be safely skipped on first reading. A more detailed description of the approach is given in [Oppitz99].

5.3.1 Core Translation

In this section we give an overview of the basic approach taken, describing how synchronization is implemented.

Every VDM++ class that is translated has the following included in its Java translation:

- An `evaluatePP` method

- An inner `Sentinel` class and a Sentinel member variable named `sentinel`

Of course, these are all in addition to the existing instance variables, and functions of the VDM++ class that are translated in the manner described in the preceding chapters. Operations are translated largely as before, but with one minor adjustment described below. We now briefly describe each of the Java components listed above.

The `evaluatePP` method is specified by the `EvaluatePP` interface in the Concurrent VDM Java Library which each translated class implements. It takes as argument an integer representing the name of one of the operations from that VDM++ class, and returns true or false corresponding to the evaluation of the permission predicate for that operation (identically true if no permission predicate exists for that operation).

The `Sentinel` class is used to record history counter information. An operation *Op* in the VDM++ class will be translated using the following schema

```
sentinel.entering(((Op Sentinel) sentinel).Op);
try {
  Translation of body of op
}
finally { sentinel.leaving(((Op Sentinel) sentinel).Op);}
```

The call to `sentinel.entering` updates the `#req` history counter and then evaluates the permission predicate for the operation using the `evaluatePP` method. If the permission predicate evaluates to true, the call to `sentinel.entering` finishes and the body executes; otherwise the call blocks, waiting to be notified of any activity with respect to history counters. Note that there is no notification of any change in the value of any member variables corresponding to instance variables, even though these may be used in other permission predicates. However, this just mirrors the semantics of VDM++ which does not require re-evaluation of permission predicates when instance variables are altered.

Similarly at the end of the operation there is a call to `sentinel.leaving` that updates the appropriate history counters. This is enclosed within a `finally` statement to ensure that it is executed whether the body terminates normally or abnormally.

As well as these additions a couple of modifications are also made to the translation strategy:

- VDM++ instance variables are translated into Java *volatile* member variables, since they might be shared amongst several threads.

- The class constructor is extended to initialize the sentinel.

5.3.2   Procedural Threads

If the VDM++ class to be translated contains a procedural thread the core translation is extended in four ways:

- The translated class implements the `Runnable` interface from the `java.lang` package. This is in addition to implementing the `EvaluatePP` interface.

- A `VDMThread` member variable is added; `VDMThread` is defined as part of the Concurrent VDM Java Library.

- A `run` method is implemented in the class's body, as specified by the *Runnable* interface. The body of this method corresponds to the translation of the `thread` clause in the VDM++ class.

- A `start` method is added. It initializes the thread and then starts it using the thread's own `start` method.

5.3.3   Periodic Threads

If the VDM++ class to be translated contains a periodic thread the core translation is extended in three ways:

- A `PeriodicThread` member variable called `perThread` is added. `PeriodicThread` is defined in the Concurrent VDM Java Library.

- In the constructor `perThread` is initialized and its `threadDef` method is defined to be whichever operation is specified to be executed periodically in the VDM++ class.

- A `start` method is added. It starts `perThread` using its `invoke` method.

5.4   Example

We illustrate the Concurrent VDM++ to Java Code Generator with an example of a Timer. The Timer maintains instance variables recording the current time, and has two operations: one for setting the time and one for incrementing the time. The latter operation is executed every 1000 milliseconds by the class's periodic thread.

```
class Timer

  instance variables
    hour: nat := 0;
    min: nat := 0;
    sec: nat := 0

  operations
    IncrementTime: () ==> ()
    IncrementTime() == (
        sec := sec + 1;
        if sec = 60 then (sec := 0; min := min + 1);
        if min = 60 then (min := 0; hour := hour + 1);
        if hour = 24 then hour := 0;
    );

    -- This is for use by threads other than the periodic thread
    public SetClock: nat * nat * nat ==> ()
    SetClock(h,m,s) == (
      hour := h;
      min  := m;
      sec  := s
    );

  thread
    periodic (1000) (IncrementTime)
```

```
  sync
    mutex(IncrementTime, SetClock);

end Timer
```

Note that *IncrementTime* and *SetClock* are mutually exclusive as they both write to the three instance variables. This is expressed in the class's *sync* clause.

The corresponding Java code is listed below. Those parts highlighted in grey are specific to the translation of concurrency constructs.

```
public class Timer implements EvaluatePP {

  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
  private volatile Integer hour = null;
  private volatile Integer min = null;
  private volatile Integer sec = null;
  volatile Sentinel sentinel;
  PeriodicThread perThread;

  class TimerSentinel extends Sentinel {

    public final int IncrementTime = 0;
    public final int SetClock = 1;
    public final int nr_functions = 2;

    public TimerSentinel () throws CGException{}

    public TimerSentinel (EvaluatePP instance) throws CGException{
      init(nr_functions, instance);
    }
  };

  public Boolean evaluatePP (int fnr) throws CGException{
    Boolean temp;

    switch(fnr) {
    case 0: {
      temp = new Boolean(UTIL.equals(
               new Integer(sentinel.active[((TimerSentinel) sentinel).IncrementTime]
                          + sentinel.active[((TimerSentinel) sentinel).SetClock]),
               new Integer(0)));
      return temp;
    }    case 1: {
      temp = new Boolean(UTIL.equals(
               new Integer(sentinel.active[((TimerSentinel) sentinel).IncrementTime]
                          + sentinel.active[((TimerSentinel) sentinel).SetClock]),
               new Integer(0)));
      return temp;
    }
    }
    return new Boolean(true);
  }

  public void setSentinel () {
    try{
      sentinel = new TimerSentinel(this);
    }
    catch (CGException e) {
      System.out.println(e.getMessage());
    }
  }
```

```
public void start () throws CGException{
  perThread.invoke();
}

public Timer () {
  try{
    perThread = new PeriodicThread(new Integer(1000),perThread){

      public void threadDef () throws CGException{
        IncrementTime();
      }
    };
    setSentinel();
    hour = new Integer(0);
    min = new Integer(0);
    sec = new Integer(0);
  }
  catch (Throwable e) {
    System.out.println(e.getMessage());
  }
}

private void IncrementTime () throws CGException{
  sentinel.entering(((TimerSentinel) sentinel).IncrementTime);
  try{
    sec = UTIL.NumberToInt(UTIL.clone(new Integer(sec.intValue() +
                                            new Integer(1).intValue())));
    if (new Boolean(sec.intValue() == new Integer(60).intValue()).booleanValue()) {
      sec = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
      min = UTIL.NumberToInt(UTIL.clone(new Integer(min.intValue() +
                                              new Integer(1).intValue())));
    }
    if (new Boolean(min.intValue() == new Integer(60).intValue()).booleanValue()) {
      min = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
      hour = UTIL.NumberToInt(UTIL.clone(new Integer(hour.intValue() +
                                              new Integer(1).intValue())));
    }
    if (new Boolean(hour.intValue() == new Integer(24).intValue()).booleanValue())
      hour = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
  }
  finally {
    sentinel.leaving(((TimerSentinel) sentinel).IncrementTime);
  }
}

public void SetClock (final Integer h, final Integer m, final Integer s) throws CGException{
  sentinel.entering(((TimerSentinel) sentinel).SetClock);
  try{
    hour = UTIL.NumberToInt(UTIL.clone(h));
    min = UTIL.NumberToInt(UTIL.clone(m));
    sec = UTIL.NumberToInt(UTIL.clone(s));
  }
  finally {
    sentinel.leaving(((TimerSentinel) sentinel).SetClock);
  }
}
};
```

5.5 Limitations

When using the Concurrent VDM++ to Java Code Generator the following should be taken into account:

- In general, Java classes generated by the sequential code generator may only be used in concurrent systems in an unsynchronized manner since the synchronization mechanism is an integral part of the translated classes rather than an adjunct. If synchronization is required then the code should be regenerated using the Code Generator with the concurrency option.

- For periodic threads, it is the specifier's responsibility to ensure that the execution time of the operation to be executed periodically is less than the period. Failure to do so could result in uncaught exceptions.

- The `startlist` statement is not currently supported by the Concurrent VDM++ to Java Code Generator.

A   References

[Gosling&00]   James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Langauge Specification, Second Edition. The Java Series*, Addison Wesley, 2000.

[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.

[Oppitz99]   Oliver Oppitz. *Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox.* Master's thesis, TU Graz, Austria, April 1999. 151 pages.

[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   Installing the Code Generator

The VDM++ to Java Code Generator is an add-on feature to the VDM++ Toolbox. Its installation is described in [InstallPPMan]. You will find a directory named

    javacg

in its distribution. This directory contains 1 file and 2 other directories:

- `VDM.jar`: the VDM Java Library (see Appendix C).

- `libdoc`: containing HTML documentation of the VDM Java Library (see Appendix C).

- `example`: containing the `DoSort` example used to illustrate the Code Generator in this manual (see Appendix D).

C   The VDM Java Library

As described in Section 2.2 you must ensure that your `CLASSPATH` environment variable includes the VDM Java Library, i.e., the `VDM.jar` file, in order to be able to run the generated code. This file can be found in

        `javacg/`

Moreover, the

        `javacg/libdoc`

directory contains HTML documention generated by `javadoc` of this library.

D The DoSort Example

The `DoSort` example, which is used to illustrate the Code Generator in this manual, can be found in the directory named `javacg/example`. The directory contains the following files:

- `Sort.rtf`
- `sort.vpp`
- `DoSort.java`
- `MainSort.java`

The java files can be compiled by executing the following command in the `javacg/example` directory:

```
javac -classpath ../VDM.jar DoSort.java MainSort.java
```

If you are using the Unix Bourne shell or a compatible shell, the main program can be run be executing the following command:

```
java -classpath .:../VDM.jar MainSort
```

If you are working on a Windows-based system, you must use ";" and not ":" as the delimiter:

```
java -classpath .;../VDM.jar MainSort
```

D.1 VDM+++ Specification of Class `DoSort` (`Sort.rtf`)

```
class DoSort

operations
  public Sort: seq of int ==> seq of int
  Sort(l) ==
    return DoSorting(l)

functions
```

```
  protected DoSorting: seq of int -> seq of int
  DoSorting(l) ==
    if l = [] then
      []
    else
      let sorted = DoSorting (tl l) in
        InsertSorted (hd l, sorted);

  private InsertSorted: int * seq of int -> seq of int
  InsertSorted(i,l) ==
    cases true :
      (l = [])    -> [i],
      (i <= hd l) -> [i] ^ l,
      others      -> [hd l] ^ InsertSorted(i,tl l)
    end

end DoSort
```

D.2   Java Code of Class `DoSort` (`DoSort.java`)

```
//
// THIS FILE IS AUTOMATICALLY GENERATED!!
//
// Generated at Fri 20-Oct-2000 by the VDM++ JAVA Code Generator
// (v6.6 - Fri 20-Oct-2000)
//
// Supported compilers:
// jdk1.3
//

// ***** VDMTOOLS START Name=HeaderComment KEEP=NO

// ***** VDMTOOLS END Name=HeaderComment

// ***** VDMTOOLS START Name=package KEEP=NO
// ***** VDMTOOLS END Name=package


// ***** VDMTOOLS START Name=imports KEEP=NO

import dk.ifad.toolbox.VDM.*;
import java.util.*;
// ***** VDMTOOLS END Name=imports
```

```
public class DoSort implements EvaluatePP {

// ***** VDMTOOLS START Name=vdmComp KEEP=NO
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp

// ***** VDMTOOLS START Name=sentinel KEEP=NO
  volatile Sentinel sentinel;
// ***** VDMTOOLS END Name=sentinel


// ***** VDMTOOLS START Name=DoSortSentinel KEEP=NO
  class DoSortSentinel extends Sentinel {

    public final int Sort = 0;

    public final int nr_functions = 1;


    public DoSortSentinel () throws CGException{}


    public DoSortSentinel (EvaluatePP instance) throws CGException{
      init(nr_functions, instance);
    }

  }
// ***** VDMTOOLS END Name=DoSortSentinel
;

// ***** VDMTOOLS START Name=evaluatePP KEEP=NO
  public Boolean evaluatePP (int fnr) throws CGException{
    return new Boolean(true);
  }
// ***** VDMTOOLS END Name=evaluatePP


// ***** VDMTOOLS START Name=setSentinel KEEP=NO
  public void setSentinel () {
    try{
      sentinel = new DoSortSentinel(this);
    }
    catch (CGException e) {
      System.out.println(e.getMessage());
    }
  }
// ***** VDMTOOLS END Name=setSentinel


// ***** VDMTOOLS START Name=DoSort KEEP=NO
```

```
  public DoSort () {
    try{
      setSentinel();
    }
    catch (Throwable e) {
      System.out.println(e.getMessage());
    }
  }
// ***** VDMTOOLS END Name=DoSort



// ***** VDMTOOLS START Name=Sort KEEP=NO
  public Vector Sort (final Vector l) throws CGException{
    sentinel.entering(((DoSortSentinel) sentinel).Sort);
    try{
      return (Vector) UTIL.ConvertToList(DoSorting(l));
    }
    finally {
      sentinel.leaving(((DoSortSentinel) sentinel).Sort);
    }
  }
// ***** VDMTOOLS END Name=Sort



// ***** VDMTOOLS START Name=DoSorting KEEP=NO
  protected Vector DoSorting (final Vector l) throws CGException{
    Vector varRes_3 = null;

    if (new Boolean(UTIL.equals(l, new Vector())).booleanValue())
      varRes_3 = new Vector();
    else {
      Vector sorted = (Vector) UTIL.ConvertToList(DoSorting(
                                 new Vector(l.subList(1, l.size()))));

      varRes_3 = (Vector) UTIL.ConvertToList(InsertSorted(
                                 UTIL.NumberToInt(l.get(0)), sorted));
    }
    return varRes_3;
  }
// ***** VDMTOOLS END Name=DoSorting



// ***** VDMTOOLS START Name=InsertSorted KEEP=NO
  private Vector InsertSorted (final Integer i, final Vector l)
      throws CGException{
    Vector varRes_4 = null;

    boolean succ_5 = true;

    {
```

```
      succ_5 = true;
      if (!UTIL.equals(new Boolean(true), new Boolean(UTIL.equals(l,
                                           new Vector())))))
        succ_5 = false;
      if (succ_5) {
        varRes_4 = new Vector();
        varRes_4.add(i);
      }
    }
    if (!succ_5) {
      succ_5 = true;
      if (!UTIL.equals(new Boolean(true), new Boolean((i.intValue())
             <= (UTIL.NumberToInt(l.get(0)).intValue())))))
        succ_5 = false;
      if (succ_5) {
        Vector var1_16 = null;

        var1_16 = new Vector();
        var1_16.add(i);
        varRes_4 = (Vector) var1_16.clone();
        varRes_4.addAll(l);
      }
    }
    if (!succ_5) {
      Vector var1_19 = null;

      var1_19 = new Vector();
      var1_19.add(UTIL.NumberToInt(l.get(0)));
      varRes_4 = (Vector) var1_19.clone();
      varRes_4.addAll((Vector) UTIL.ConvertToList(InsertSorted(i,
                            new Vector(l.subList(1, l.size())))));
    }
    return varRes_4;
  }
// ***** VDMTOOLS END Name=InsertSorted


}
;
```

D.3   The Handcoded Java Main Program (`MainSort.java`)

```
import dk.ifad.toolbox.VDM.*;
import java.util.*;

public class MainSort {

  public static void main(String[] args){
```

```
    try{
    Vector arr = null;
    arr = new Vector();
    arr.add(new Integer(23));
    arr.add(new Integer(1));
    arr.add(new Integer(42));
    arr.add(new Integer(31));
    DoSort dos = new DoSort();
    System.out.println("Evaluating Sort("+UTIL.toString(arr)+"):");
    Vector res = dos.Sort(arr);
    System.out.println(UTIL.toString(res));
    }
    catch (CGException e){
      System.out.println(e.getMessage());
    }
  }
}
```