SCSK

VDMTools

The VDM Toolbox API

SCSK CORPORATION

How to contact SCSK:

http://www.vdmtools.jp/	VDM information web site(in Japanese)
http://www.vdmtools.jp/en/	VDM information web site(in English)
http://www.scsk.jp/	SCSK Corporation web site(in Japanese)
http://www.scsk.jp/index_en.html	SCSK Corporation web site(in English)
vdm.sp@scsk.jp	Mail

The VDM Toolbox API 2.0
— Revised for VDMTools v9.0.2

Contents

1 Introduction 1

2 CORBA - The Basics 2
2.1 IDL 2

3 The VDM Toolbox API 3
3.1 IDL Description of The Tool API 3
3.1.1 VDMProject 4
3.1.2 VDMModuleRepos 4
3.1.3 VDMParser 5
3.1.4 VDMInterpreter 6
3.1.5 VDMErrors 8
3.2 IDL Description of VDM Values 9
3.2.1 VDM Values as Distributed Objects 11
3.2.2 Using Values Returned from the Interpreter 12
3.2.3 Constructing VDM Values in the Client 13
3.2.4 Converting Distributed VDM Values to “real” VDM C++ Values 14
3.3 Handling of Exceptions 15

4 Writing a C++ Client 16
4.1 Choosing a CORBA Implementation 16
4.2 Implementing a Client 17
4.2.1 Initializing the CORBA Services 17
4.2.2 Acquiring the Application Object 18
4.2.3 Object References in C++ 20
4.2.4 Configuring the Current Project 20
4.2.5 Using the Parser 21
4.2.6 Using the Type Checker 22
4.2.7 Using the Interpreter 22
4.2.8 Additional Aspects of the Example 24
4.3 Compiling the Client 24
4.3.1 Supported Compilers 25
4.4 Running the Client 25

5 Writing a Java Client 26
5.1 Choosing a CORBA Implementation 26
5.2 Implementing a Client 27
5.2.1 Importing CORBA Services 27
5.2.2 Acquiring the Application Object 27

5.2.3 Configuring the Current Project 30
5.2.4 Using the Parser 30
5.2.5 Using the Type Checker 31
5.2.6 Using the Interpreter 32
5.2.7 Additional Aspects of the Example 34
5.3 Compiling the Client 34
5.4 Running the Client 35

6 API Reference Guide **36**
6.1 Corba API 36
6.1.1 Types 36
6.1.2 Error Structure 36
6.1.3 ModuleStatus Structure 37
6.1.4 VDMApplication Interface 37
6.1.5 VDMCodeGenerator Interface 38
6.1.6 VDMErrors Interface 39
6.1.7 VDMInterpreter Interface 39
6.1.8 VDMModuleRepos Interface 43
6.1.9 VDMParser Interface 44
6.1.10 VDMPrettyPrinter Interface 45
6.1.11 VDMProject Interface 45
6.1.12 VDMTypeChecker Interface 46
6.2 VDM API 46
6.2.1 Types 47
6.2.2 VDM::VDMGeneric Interface 47
6.2.3 Basic VDM Types 48
6.2.4 VDM::VDMMap Interface 49
6.2.5 VDM::VDMRecord Interface 50
6.2.6 VDM::VDMSequence Interface 50
6.2.7 VDM::VDMSet Interface 51
6.2.8 VDMTuple Interface 52
6.2.9 VDMFactory Interface 53
6.3 Exceptions 53
6.4 C++ API Reference 54
6.4.1 corba_client.h 54
6.4.2 Naming Conventions 56
6.4.3 Casting 56
6.5 Java API 56
6.5.1 Helper Classes 57
6.5.2 Holder Classes 58

7 Recommended Reading **59**

A Example Programs **60**

A.1 The C++ Client Example 60

A.2 The Java Client Example 69

1 Introduction

This document describes how to use the CORBA-based API of the VDM Toolbox.

The VDM Toolbox API allows you to write programs (clients) that access and modify certain properties of a running instance of a VDM Toolbox (graphical or command line). The VDM Toolbox can be accessed by several CORBA clients at the same time. These clients can - through the API - access and configure the project, parse and type check individual files, evaluate expressions through the interpreter, etc. The client processes and the VDM Toolbox are separate processes that may run on different machines, possibly running on different operating systems, on the network. As a consequence, a VDM Toolbox being used as server by any client process is also available to the user through its user interface.

The API is based on CORBA (see [OMG96]). For this reason the API is accessible from any language for which there exists a CORBA 2.0 compliant implementation. For example you could easily write your client in either C++ or Java, since several (free) CORBA implementations are available for these languages. Throughout Section 3 small pieces of example code written in C++ are provided. In sections 4 and 5 however, we will describe how to write a complete client in C++ and Java respectively.

This document and the API it describes apply to both the VDM-SL and VDM++ version of the Toolbox. The API only differentiates between VDM-SL and VDM++ in a few cases, and they are explicitly stated in the definition of the API. In general, when we use the term “module” in this manual and in the definition of the API, we refer to either a module in VDM-SL or a class in VDM++.

2 CORBA - The Basics

The main idea in CORBA is *distribution of objects*. A client process can create, access and possibly modify the state of objects handled by and physically contained in a separate server process located locally or remotely on the network. The client has “a handle” to the object contained in the server and it uses this handle to make method calls, as if the distributed object is located in the address space of the client. The CORBA standard specifies how a handle to a distributed object can be acquired as well as how methods are invoked, and values are passed between different objects.

Since CORBA is only a standard for object distribution, an implementation of CORBA (a so-called *ORB*) is necessary to write CORBA servers and clients. Currently CORBA implementations are available for a multitude of different platforms and languages.

2.1 IDL

The objects exposed by a CORBA enabled server are described using the Interface Definition Language (IDL). IDL is an object-oriented language for describing interfaces in an implementation language and in a platform neutral way. Vendors that provide tools with a CORBA interface make the interface known to clients by distributing the IDL description with the tool. The syntax of IDL is described in [OMG96].

When implementing a client, the IDL description is mapped to the preferred implementation language using an IDL compiler (comes with the chosen CORBA implementation). The code generated from the IDL description is compiled and linked with the client executable making it capable of using the CORBA interface of the server.

3 The VDM Toolbox API

The CORBA interface of the VDM Toolbox is described in the two IDL files `corba_api.idl` and `metaiv_idl.idl`, that are also distributed with the VDM Toolbox. The first file describes the actual interface of the VDM Toolbox whereas the second file describes the interface of different VDM values that can be passed between a client and the VDM Toolbox. In the following both files will be described in detail, and in Section 6 a reference manual for these interfaces is provided.

3.1 IDL Description of The Tool API

The API of the VDM Toolbox consists of a number of different objects (interfaces in IDL) accessible from a client process. The object `VDMApplication`, from which all other aspects of the API are available, is the main entry point. This object is the client's handle to the VDM Toolbox, and must consequently be constructed prior to using any other functionality of the API. In section 4 and Section 5 we will describe how to acquire this handle to the VDM Toolbox in C++ and Java respectively.

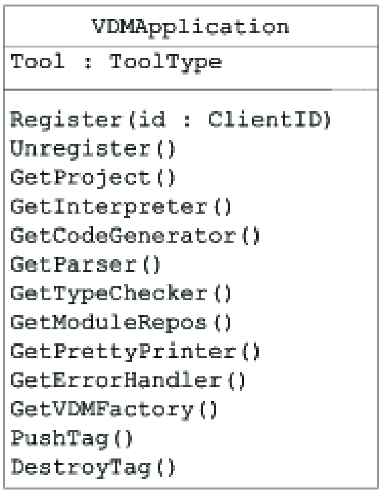

Figure 1: The `VDMApplication` interface.

The `VDMApplication` interface is shown in Figure 1. The methods `Register` and `Unregister` are used by a client to register and unregister its process at the server. Moreover, the `VDMApplication` interface consists of a number of methods returning other interfaces. For instance, if you wish to configure the current project of the VDM Toolbox, use `GetProject` to get a handle to the project interface, that is, a handle to the `VDMProject` interface described below. Additionally the `Tool` attribute of the interface can be used to decide the type of tool used as server, i.e. whether the client is connected to a VDM-SL or a VDM++ Toolbox. For detailed information on how to read and modify the value of attributes see Sections 4 and 5.

3.1.1 VDMProject

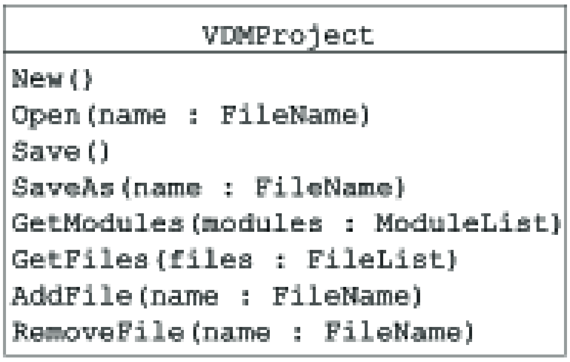

Figure 2: The `VDMProject` interface.

The `VDMProject` interface is shown in Figure 2. Using this interface it is possible to access and modify the current project of the VDM Toolbox. `GetFiles` and `GetModules` return (through a parameter) a sequence of file names and module names in the current project. `AddFile` and `RemoveFile` are used to configure the project.

3.1.2 VDMModuleRepos

The `VDMModuleRepos` interface is shown in Figure 3.

The interface `VDMModuleRepos` is used to acquire additional information on a given module or class. `FilesOfModule` returns the files of a particular module, while `Status` retrieves the current status, as indicated by the S, T, C, and P indicators in the user interface, of a given module. The four remaining methods are only available from the VDM++ Toolbox. They are used to query the inheritance and association relationships of a class. Use these methods to find the

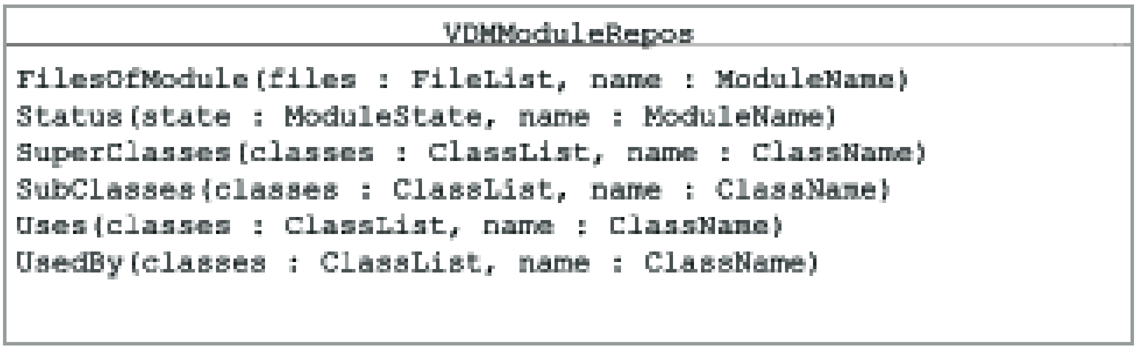

Figure 3: The `VDMModuleRepos` interface.

super or sub classes of a class as well as to find out how classes reference each other.

Notice that, since this IDL description is common to both the VDM++ and VDM-SL Toolbox, whenever we use *ModuleName* or *ModuleList* in the definitions this applies to both modules (from VDM-SL) and classes (from VDM++). However if *ClassName* or *ClassList* is explicitly used, application is restricted to VDM++ only.

3.1.3 VDMParser

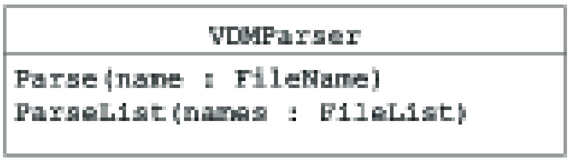

Figure 4: The `VDMParser` interface.

The `VDMParser` interface is shown in Figure 4. The interface can be used to have the VDM Toolbox parse either a single file or a list of files. In the latter case the file list will normally have been acquired by a call to `VDMProject::GetFiles`, instead of manually constructing the list.

If errors are encountered while parsing the file(s) the `VDMErrors` interface (described in Section 3.1.5) can subsequently be queried to gain detailed information describing the errors detected.

The structure of the interfaces for the type checker, code generator and pretty printer (VDMCodeGenerator, VDMTypeChecker and VDMPrettyPrinter) are very similar to VDMParser, with the only difference that these interfaces have a number of different attributes that can be read and modified from the client. The setting of such attributes control the functionality of the particular interface. These three interfaces will not be described in detail here, and we refer to the IDL description in Section 6 for further details and descriptions of the individual attributes.

3.1.4 VDMInterpreter

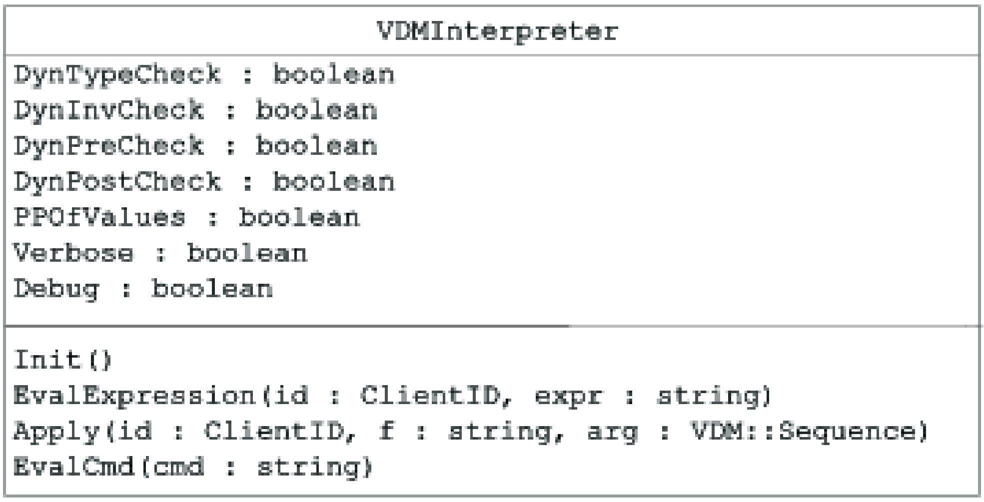

Figure 5: The `VDMInterpreter` interface.

The `VDMInterpreter` interface is shown in Figure 5. This interface allows you to use the interpreter to evaluate and debug VDM expressions and invoke functions and operations in the specification. Calling `EvalExpression(client_id, expr)` will evaluate the expression in the string argument *expr* and return the result to the client. The result will be represented as the VDM value `Generic` described in Section 3.2. For instance,

```
EvalExpression(client_id, "[e | e in set {1,...,20}
                & exists1 x in set {2,...,e} & e mod x = 0 ] ")
```

would return a `Generic` holding the sequence of all primes between one and twenty. Alternatively one could specify (in VDM) a more efficient function, `Primes`, for extracting all primes from a sequence and invoke it through the `Apply` method of the interface:

```
Apply(client_id, "Primes", s)
```

with `s` being the argument for the function. `Apply` will also return the result of applying the function to the given arguments as a VDM value contained in an `Generic`. (In fact this example is a slight simplification of how to pass arguments for a function when using `Apply`. We will describe the correct way to use `Apply` in Section 4.2.7 for C++ clients, and Section 5.2.6 for Java clients.

In this example it is convenient to use the interpreter to construct the sequence of integers:

```
s = EvalExpression(client_id, "[e|e in set {1,...,20}]")
```

and use the returned value `s` as argument to `Apply`. Alternatively the client could have manually constructed the sequence.

Apart from the functions already mentioned, the interpreter interface holds a number of attributes (boolean values) that can be modified from the client. The settings of these attributes control the way the interpreter behaves. The first five attributes (`DynTypeCheck`, `DynInvTheck`, `DynPreCheck`, `DynPostCheck`, `PPOfValues`) corresponds to the options for the interpreter that it is possible to set from the user interface of the VDM Toolbox. They control aspects such as dynamic type checking of invariants, pre- and post conditions etc. The two remaining attributes, `Verbose` and `Debug`, control how the API uses the interpreter. `Verbose` controls whether or not the result of using the interpreter should be echoed in the user interface of the VDM Toolbox. If `Verbose` is false the client will use the interpreter "silently" without echoing results to the user interface. The attribute `Debug` controls whether breakpoints in the specification are respected during evaluation or not. If `Debug` is set to true the evaluation will be suspended at each breakpoint and the user is able to debug the specification. In this case the call to `Apply` and `EvalExpression` will not return before the user has finished the debugging.

It is possible to set breakpoints using the methods `SetBreakPointByPos` and `SetBreakPointByName`. While the first method takes a file and a position (line, column) as parameters, the latter expects the name of the module and a function name. Both methods return the number of the breakpoint that has been set. This number can be used to delete the breakpoint again (`DeleteBreakPoint`).

Debugging is then started by calling `StartDebugging`. The method takes the `ClientID` and an expression (a string) as parameter. `StartDebugging` returns, when the evaluation is finished or a breakpoint has been encountered. It returns a `VDMTuple`, containing the evaluation state (either `<BREAKPOINT>`, `<INTERRUPT>`, `<SUCCESS>` or `<ERROR>`) and, in case of `<SUCCESS>`, the result of the evaluation as a MetaIV value. The methods `DebugStep`, `DebugStepIn`, `DebugSingleStep` and `DebugContinue` can be used to step through the specification.

Assume we have a module `A` that contains two functions:

```
module A
  ...
  functions
     foo: nat -> nat
     foo (a) == a + 1;

     bar: nat -> nat
     bar (b) = foo (b)
  ...
```

We could use `SetBreakPointByName ("A", "foo")` to set a breakpoint for the function `foo`. The call of `StartDebugging (id, "A‘bar(1))` would then return after the call of `foo (b)` in `bar` has been encountered. The result would be `mk_( <BREAKPOINT>,<<UNDEFINED>> )`. A call to `DebugContinue` continues the evaluation and would return `mk_( <SUCCESS>, 2 )`.

3.1.5 VDMErrors

The `VDMErrors` interface is shown in Figure 6. The state of this interface is updated if errors are encountered during parsing, type checking, code generation or pretty printing. Use this interface to query the number of errors and/or warnings through the attributes `n_err` and `n_warn`. The two methods of the interface return a sequence of error or warning descriptors used to gain detailed information.

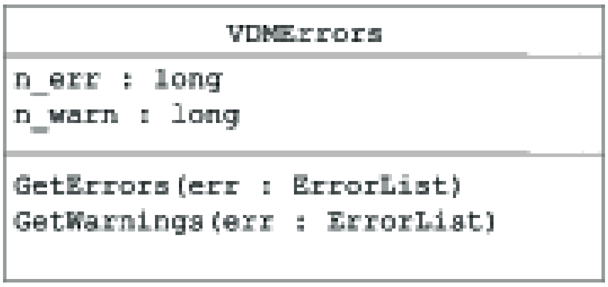

Figure 6: The `VDMErrors` interface.

3.2 IDL Description of VDM Values

Having described the interface of the VDM Toolbox, we will now proceed with a description of how VDM values are passed through the API, i.e. how VDM values can be passed from the VDM Toolbox to the client and vice versa.

The given code examples are written in C++. We refer to section 5 for the Java syntax.

As already mentioned, the `EvalExpression` method of the `VDMInterpreter` interface returns the result of the evaluation as a VDM value, and the `Apply` method takes as argument a VDM sequence of VDM values as the list of arguments for a function or operation. How to use and manipulate such VDM values is documented in Section 6.2 and also described in the IDL file `metaiv_idl.idl`. The structure of the IDL interface is kept as tight as possible to the structure of the VDM C++ Library (as described in [SCSa]). Each class of the VDM C++ Library corresponds to an interface (with the same name) in the IDL description. Figure 7 illustrates the fact that all concrete VDM values inherit from the same super class, `Generic`. Notice that the figure only shows a subset of the available VDM values.

The following is an example of how to read the contents of a VDM value returned from the interpreter:

```
VDM::VDMGeneric_var g;
g = interp->EvalExpression(client_id,
                           "{ e |-> 2**e | e in set {1,...,16}}");
if(!g->IsMap()){
  // signal an error...
  ...
```

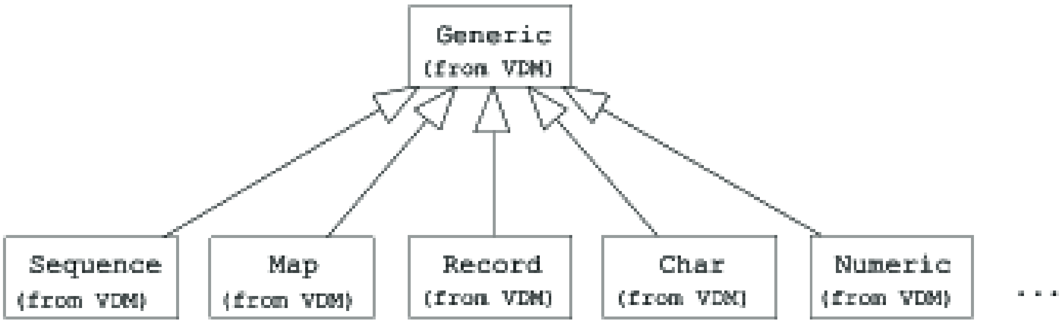

Figure 7: Inheritance structure of VDM values.

```
06  }
07  else{
08    VDM::VDMMap_var m;
09    m = VDM::VDMMap::_narrow(g);
10    VDM::VDMGeneric_var iter;
11    for(int i = m->First(iter); i; i = m->Next(iter)){
12      VDM::VDMGeneric_var rng = m->Apply(client_id, iter);
13      cout << iter->ToAscii() << "-->" << rng->ToAscii() << "\n";
14      iter->Destroy();
15      rng->Destroy();
16    }
17  }
18  g->Destroy();
```

The interpreter returns the result of an evaluation in a `VDMGeneric`. For this reason the variable `g` is declared as a `VDM::Generic_var`[1], and used to hold the result from `EvalExpression`. The expression evaluated by the interpreter is a map comprehension; hence the value contained in `g` should be of type `Map`. The method `IsMap` of the `Generic` interface can be used to check that this is indeed the case, as seen in line 3. If `g` is not of type `Map` an error is signalled. Otherwise it is safe to convert the Generic to a Map type. The way to cast (or narrow) an object reference is to use the `_narrow` method supplied by the ORB implementation. Line 9 shows how to narrow from `VDMGeneric` to `VDMMap`. With an object reference, `m`, of type `VDMMap` all the methods of the Map interface are now available. Using `First` and `Next` it is possible to iterate through the domain of the map (line 11), and with `Apply` (line 12) the value associated with a key

[1]See Section 4 for a description of the special `_var` types used in the client and how to use CORBA object references.

in the map can be retrieved. Keep in mind, that only *one* Client should access these methods at a time. If you use them concurrently, the clients may not get all values contained by the `VDMMap`.

To facilitate the printing of VDM values, the Generic interface (and hence all other VDM values) provides the `ascii` method returning the ASCII representation of the VDM value. In line 13 this method is used to print out each element of the map `m`.

To summarize: This simple example uses the interpreter to construct a map that is subsequently printed using the various methods of `VDMMap` to iterate through the map. The output from the example above is:

```
1-->2
2-->4
3-->8
... (lines removed for brevity) ...
16-->65536
```

3.2.1 VDM Values as Distributed Objects

Whenever VDM values are passed from the server to the client they are passed as "handles" or object references to distributed objects contained in the server. That is, the real VDM values used by a client are actually managed by the VDM Toolbox and are contained in the address space of the server. For this reason *all* VDM values held by a client must be explicitly freed in the server when the client will not use the value any longer. The client does this by calling the `Destroy` method of the VDM value. In lines 14 and 15, of the example above, the object `iter`, used to represent each element in the domain of `m`, and `rng`, used to hold the corresponding element of the range of `m`, are destroyed. Finally, in line 18 the VDM value created by the interpreter, `g`, is destroyed. If values are not destroyed in this way when the client does not need them any more, they will never be released in the VDM Toolbox. As a consequence the VDM Toolbox process will use an increasing amount of memory.

An alternative to explicitly destroy objects by calling the `Destroy` method on the object is to use the two methods `PushTag` and `DestroyTag` of the `VDMApplication` interface. Calling `PushTag` will generate a unique tag and push it onto an internal tag stack. The tag stored on top of the tag stack is used to tag all objects subsequently created by the VDM Toolbox. Each call to `DestroyTag` will pop the topmost tag of the tag stack and call destroy on each object tagged with this value. As a consequence all objects created since the last call to `PushTag` will be destroyed. Using the combination of `PushTag` and `DestroyTag` the previous example now reads as follows:

```
app->PushTag(client_id);
VDM::VDMGeneric_var g;
g = interp->EvalExpression(client_id,
                        "{ e |-> 2**e | e in set {1,...,16}}");
if(!g->IsMap()){
  // signal an error...
  ...
}
else{
  VDM::VDMMap_var m;
  m = VDM::VDMMap::_narrow(g);
  VDM::VDMGeneric_var iter;
  for(int i = m->First(iter); i; i = m->Next(iter)){
    VDM::VDMGeneric_var rng = m->Apply(client_id, iter);
    cout << iter->ToAscii() << "-->" << rng->ToAscii() << "\n";
  }
}
app->DestroyTag(client_id);
             // All objects created since last PushTag()
             // will now be destroyed.
```

Notice that calls to `PushTag` and `DestroyTag` can be nested to any depth as long as the total number of calls to `DestroyTag` does not exceed the total number of calls to `PushTag`.

3.2.2 Using Values Returned from the Interpreter

In the example above we used the `ToAscii` method to print the range and domain of the map generated by the interpreter. The value, as opposed to the ASCII representation, is available through the `GetValue` method. For instance, the following code squares each element of a sequence and prints out the result:

```
app->PushTag(client_id);
g = interp->EvalExpression(client_id,
                           "[ e | e in set {1,...,10}]");
```

```
if(!g->IsSequence()){
  exit(-1);
}
else{
  VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
  for(int i = 1; i <= s->Length(); i++){
    VDM::VDMGeneric_var e = s->Index(i);
    if(e->IsNumeric()){
      VDM::VDMNumeric_var ii = VDM::VDMNumeric::_narrow(e);
      cout << ii->GetValue() * ii->GetValue() << " ";
    }
  }
}
app->DestroyTag(client_id);
```

Notice that the `PushTag` and `DestroyTag` methods are also used in this example to make sure that values used by the client are released in the VDM Toolbox when the client no longer needs them.

In what follows we will assume that all code examples are “wrapped” with calls to `PushTag` and `DestroyTag`, so that we will not have to call `Destroy` explicitly.

3.2.3 Constructing VDM Values in the Client

The VDM values we have used so far were all created by the interpreter and returned to the client. However, the client can construct VDM values directly by using the `VDMFactory` interface. The following example acquires a handle to the `VDMFactory` interface and constructs a set of numeric values:

```
VDM::VDMFactory_var fact = app->GetVDMFactory();
VDM::VDMSet_var s = fact->MkSet(client_id);
VDM::VDMNumeric_var elem;
for(int j=0; j<20; j++){
  elem = fact->MkNumeric(client_id, j);
  s->Insert(elem);
}
```

Notice that the factory interface is used to construct the overall set as well as each numeric value to be inserted in the set. Also notice that each numeric value constructed by the factory is destroyed after it has been inserted in the set, because elements are implicitly copied when inserted into composite types such as VDMSequence, VDMSet, VDMMap, VDMRecord and VDMTuple.

3.2.4 Converting Distributed VDM Values to "real" VDM C++ Values

It is important to keep in mind that when the client receives a VDM value from the interpreter, it is simply holding a handle to a VDM value contained in the VDM Toolbox. Every time the client invokes a method on the VDM value this method call is mediated to the distributed VDM value in the VDM Toolbox. For this reason, when the client iterates through, for instance, a large sequence returned from the interpreter, the VDM Toolbox gets called for each element in the sequence. Of course this approach is not particularly efficient. To allow more efficient access to VDM values held by the client, you can use the function `GetCPPValue` declared in the file `corba_client.h`. This function converts the distributed VDM value to an IDL sequence of bytes, which can in turn be converted to a true VDM C++ value, provided that you include `metaiv.h` and link with the VDM library [SCSa]. For ease of use the conversion from a distributed VDM value to a VDM C++ value is available in `corba_client.h`. Simply call `GetCPPValue` and give it as argument the object reference you wish to convert:

```
#include "metaiv.h"

g = interp->EvalExpression(client_id, "[ e | e in set {1,...,10}]");
Generic cpp_g;  // The C++ Generic
cpp_g = GetCPPValue(g);
// Now the C++ value cpp_g is local to the client process and can
// be accessed efficiently.
```

The conversion from a VDM C++ value to an object reference can be achieved through the function `FromCPPValue`, also declared, as well as documented, in `corba_client.h`.

For Java clients, the conversion described above has to be done manually, and the VDM Java Library must be included in the classpath when compiling and executing the client [SCSb]. See the function `EchoPrimes2` in the Java program in Appendix A.2 for an example.

3.3 Handling of Exceptions

The IDL interface `metaiv_idl.idl` declares two CORBA exceptions `APIError` and `VDMError`. These two kinds of exceptions are used to signal to the client process if something goes wrong in the server process. `APIError` is used to signal errors that may appear while using the API of the VDM Toolbox (`corba_api.idl`) while `VDMError` is devoted to the signalling of errors in the use of VDM values. The contents of an `APIError` is a simple message string describing what went wrong, while a `VDMError` holds an integer indicating the error. The meaning of these integers is shown in Section 6.3.

The examples in Appendix A.1 and A.2 show how the client can handle such exceptions using standard exception handling in C++ and Java respectively.

4 Writing a C++ Client

We will here describe how to write a client using C++ on Windows and Linux.

4.1 Choosing a CORBA Implementation

The CORBA implementation we will use in this example is a CORBA 2 compliant ORB *omniORB3*. The implementation is developed by *The AT&T Laboratories Cambridge*, and freely available under the terms and conditions of the GNU General Public License. Several platforms and C++ compilers are supported by omniORB3 that implements a full mapping from IDL to C++. The ORB can be downloaded from:

`http://www.uk.research.att.com/omniORB/omniORB.html`

and available as pure C++ source code or pre-compiled for a handful of different platforms including Windows 2000/XP/Vista and Linux 2.4 or 2.6. If the distribution is not available for a particular platform it is possible to download the source code and build the executables and libraries.

To implement a client you must download the onmiORB3 distribution for either Win32 or Linux and install it (extract the archive). Once omniORB is installed you should add to the system path environment variable the absolute path to the binaries directory of omniORB. This directory contains various CORBA tools (the IDL compiler for instance). For the Windows 2000/XP/Vista distribution it also contains some libraries used by the client implementation at run-time. If you download the pre-compiled distribution this is all you need to do. Otherwise you must consult the installation instructions of omniORB to successfully compile the distribution.

Alternatives to omniORB would be e.g. *Orbacus* by *Object Oriented Concepts* and the *idlj* from Sun's Java IDL. Any ORB that implements the OMG CORBA 2.x specification and IIOP should be compatible with the VDM Toolbox API, but this has not been tested.

Orbacus implements a complete mapping from IDL to C++ as well as Java, so this ORB may be the choice it you wish to write your client in Java. Orbacus is available at:

`http://www.ooc.com/ob/`

Java IDL comes with an IDL to Java compiler and is available at

`http://java.sun.com`

However, the client example distributed with the VDM Toolbox (listed in Appendix A.1) is omniORB3 specific, so if OmniBroker is used, slight modifications of the client example are required.

4.2 Implementing a Client

In this Section we go through an example illustrating how to use the VDM Toolbox from a client written in C++. In the following presentation we will show excerpts from a complete example, which can be seen in its full length in Appendix A.1.

4.2.1 Initializing the CORBA Services

Before the VDM Toolbox can be accessed from the client the underlying CORBA implementation must be initialized. Different CORBA implementations are not necessarily initialized in the same way, so please notice that the CORBA initialization described here is omniORB specific. For ease of use the initialization procedure and the acquisition of the main application object are implemented in `corba_client.cc` and available by including `corba_client.h` in the client implementation. If you wish to implement the client on a different CORBA implementation it should not be too difficult to port the contents of `corba_client.cc`. You will find `corba_client.cc` and `corba_client.h` in the `api/corba` subdirectory of the VDM Toolbox distribution.

To initialize the CORBA services all you need to do is:

```
#include "corba_client.h"

main(int argc, char *argv[])
{
  init_corba(argc, argv);
  ...
}
```

4.2.2 Acquiring the Application Object

The easiest possibility to get hold of a CORBA-reference to the VDMApplication CORBA object is to use the `get_app` method that you can find in the above mentioned `corba_client.h` file. Since the implementation is omniORB-specific, this may not work with the ORB of your choice. Therefore, the COS NamingService and stringified references are supported, too.

The COS NamingService is a standardized CORBA Object Service that is used for managing object instances and their names. It maps names, that are saved in a directory hierarchy to CORBA Objects. Unlike the stringified object references it allows the client to access objects even if it doesn't share the file system with the VDM Toolbox. Therefore by using it, you gain flexibility. Its use is recommended by the Object Management Group (`http://www.omg.org`). You can find more information on the COS NamingService and CORBA Object Services in general on the OMG CORBA homepage (`http://www.corba.org`).

When you start either the VDM-SL or the VDM++-Toolbox, it will check if a COS NameService is running. The ORB will search for a configuration file. You can specify the location of this file using the `OMNIORB_CONFIG` environment variable (refer to the omniORB-documentation to see how to use the registry for this if you use Windows). A typical `omniORB.cfg` file will contain following entries:

```
ORBInitialHost gandalf
ORBInitialPort 2809
```

This means, that the NamingService is running on a host called `gandalf` at port 2809.

omniORB provides such a NameService (there should be an executable called `omniNames` that is part of the omniORB-distribution), but you can use virtually any other CORBA-compliant NameService as long as you make it known to the omniORB using the `omniORB.cfg` file. Please refer to the omniORB documentation for further details. The VDM-SL Toolbox binds its `VDMApplication` object to the name `SL_TOOLBOX`, of kind `VDMApplication`, while Application object of the VDM++ Toolbox uses the name `PP_TOOLBOX`, of kind `VDMApplication`. This makes it possible for the client to distinguish the objects, so that it is no problem to run an instance of each Toolbox at the same time.

Take care that you do not run two instances of the same Toolbox, because then only the `VDMApplication` object of the Toolbox that has been started first will be accessible for the client. It is no problem to run more than one client using the same VDMApplication object, but keep in mind that they will influence each other.

Another approach used to acquire the main handle to the VDM Toolbox - the VDMApplication CORBA object - is to let the client read a stringified object reference (created by the most recently started VDM Toolbox) and convert this to a CORBA object reference. All ORB implementations must implement the two functions object_to_string and string_to_object, used to encode and decode object references. The VDM Toolbox uses object_to_string to encode the application object as a string and writes this string to a file. Subsequently the client must read this file and use string_to_object to convert the string to an object reference. The file generated by the VDM Toolbox is named vdmref.ior for the VDM-SL Toolbox and vppref.ior for the VDM++ Toolbox. It is written in the location specified by the VDM_OBJECT_LOCATION environment variable. If the environment variable is not set, the file is located in the root of your home directory (as pointed to by $HOME) if the VDM Toolbox is running on Unix, and in your profile directory (as pointed to by %USERPROFILE%) if running on Windows 2000/XP/Vista.

The easiest way to acquire the application object from the client is to use get_app declared in corba_client.h.

```
main(int argc, char *argv[])
{
 ...
 /* set toolType to either SL_TOOLBOX or PP_TOOLBOX */
 ToolType toolType = ...;
 VDMApplication_var app;
 get_app(app, NULL, toolType);
 ...
```

This function will check first, if a COS NameService is running and if there is an object named SL_TOOLBOX of kind VDMApplication or PP_TOOLBOX, kind VDMApplication (depending on the toolType flag). If it cannot find the object via the NameService, it will will automatically search for the file that contains the IOR reference for the VDMApplication object. After the call to get_app the variable app is the main handle to the VDM Toolbox.

The client has to register itself in the server before performing any calls towards the server. Analogously, it has to unregister itself when it terminates. This is done by calling the `Register` and `Unregister` methods of the `VDMApplication` class.

```
client_id = app->Register();
...
app->Unregister(client_id);
```

We are now in the position to access services of the running VDM Toolbox.

4.2.3 Object References in C++

In C++ a handle to an object interface of the IDL description is contained in an *object reference*. Object references are named by adding `_var` to the name of the interface. This kind of object reference is termed an *object reference variable*[2]. For instance, `VDMApplication_var` is a handle (if properly initialized) to the `VDMApplication` interface of the server. Operations of an interface are called using the "arrow" (`->`) on a `_var` object reference, e.g. `app->GetProject()` to call the method `GetProject` of the `VDMApplication` interface `app`.

4.2.4 Configuring the Current Project

The following lines of code acquire a handle to the `VDMProject` interface of the VDM Toolbox and use the `New` and `AddFile` methods of this interface. As a result the project of the VDM Toolbox is configured to contain a single file, `sort.vdm`. Files added to the project in this way must be located in the same directory as where the VDM Toolbox was started. Otherwise the file name must be given with its absolute path. If the client tries to add a non-existing file the server will throw an exception of type `APIError` indicating the error. These exceptions are described in Section 3.3.

```
VDMProject_var prj = app->GetProject();
prj->New();  // New project
prj->AddFile("sort.vdm");
```

[2] Object references are also available in a more simple form, the `_ptr` object references. We refer to [LRG00] and [OMG96] for more information on the difference between the two types of object references. For most purposes it is sufficient to use only the `_var` object references.

4.2.5 Using the Parser

To use the parser from a client you must get a handle to the VDMParser interface, and to parse a file you call the Parse method of this interface with the file name as its single argument. For instance

```
VDMParser_var parser = app->GetParser();
parser->Parse("sort.vdm");
```

will parse the file sort.vdm.

Alternatively you could use the VDMProject interface to get the list of files configured for the current project and then parse each file of this list:

```
FileList_var fl;
prj->GetFiles(fl);

for(int i=0; i<fl->length(); i++){
  cout << (char *)fl[i] << "...Parsing...";
  if(parser->Parse(fl[i]))
    cout << "done.\n";
  else
    cout << "error.\n";
}
```

This example illustrates several important aspects of the API. Initially we declare fl to be a list of files and use GetFiles to retrieve the list of files in the current project. The type FileList is defined as an unbounded sequence of strings, as shown on page 36. Consequently the list of files, fl, has all methods of the IDL sequence as stated by the CORBA specification [OMG96]. The length of a IDL sequence can be accessed through the method length, and the individual elements can be indexed as ordinary arrays in C++.

To summarize: The above lines of code retrieve the list of files in the current project, iterate through the list, and for each item calls the Parse method to parse each file. Notice that Parse returns a boolean value indicating the success of parsing the file. Parsing all files of the project by iterating the list of files is actually more complicated than need to be. Instead you could use the ParseList method:

```
FileList_var fl;
prj->GetFiles(fl);
parser->ParseList(fl);
```

4.2.6 Using the Type Checker

The interface of the type checker is similar to the interface of the parser. The interface has a number of attributes that can be accessed and modified by the client. Attributes can be read and modified for example:

```
// Get the value of DefTypeCheck:
int dtc = tpck->DefTypeCheck();

// Set the value of ExtendedTypeCheck to true
tpck->ExtendedTypeCheck(true);
```

provided of course that `tpck` is a valid handle to the type checker interface.

4.2.7 Using the Interpreter

The following example illustrates how `EvalExpression` of the interpreter interface can be used to have the interpreter evaluate any VDM expression.

```
VDMInterpreter_var interp = app->GetInterpreter();
VDM::Generic_var g;

g = interp->EvalExpression(client_id, "[e|e in set {1,...,20} & \
  exists1 x in set {2,...,e} & e mod x = 0 ]");

if(g->IsSequence())
  cout << "All primes below 20:\n" << g->ascii() << "\n";
```

The string passed to `EvalExpression` is evaluated and the result of the evaluation is returned as a VDM value in a `VDM::Generic`, which can later be used in a call to `Apply`, or read/modified by the methods provided by the interface of the VDM values as described in Section 3.2. The backslash at the end of the line in which the call to `EvalExpression` is placed is part of the C++ syntax. It is used to indicate, that the string that contains the VDM-SL expression does not contain a linebreak (`\n`).

The following example illustrates how to use a function of a VDM specification that has been read into the VDM Toolbox:

```
interp->Init();
g = interp->EvalExpression(client_id, "MergeSort([6,4,9,7,3,42])");
```

Notice that before you can call any function of the specification you must make sure that the interpreter is initialized.

An alternative to `EvalExpression` is to use the method `Apply`, which takes as argument the name of the function or operation to apply and a sequence of arguments for the function or method. The following example creates a VDM sequence of integers to be sorted with `MergeSort`:

```
VDMFactory_var fact = app->GetVDMFactory();

VDM::Sequence_var list = fact->MkSequence(client_id);
VDM::Int_var elem;
for(int j=0; j<20; j++){
  elem = fact->MkInt(client_id, j);
  list->ImpPrepend(elem);
}
```

The resulting sequence, `list`, now contains the integers from 19 down to 0. Notice how VDM values are constructed in the client by using the `VDMFactory` interface.

To call `MergeSort` through `Apply` we have to construct the list of arguments. The arguments for the function to be called through `Apply` are contained in a `VDMSequence`. The function we want to call only takes one argument, the sequence of integers we have just constructed:

```
VDM::Sequence_var arg_l = fact->MkSequence(client_id);
arg_l->ImpAppend(list);
```

Now `MergeSort` can be applied as follows:

```
g = interp->Apply(client_id, "MergeSort", arg_l);
```

if, of course, the interpreter has been initialized. Notice that the argument list `arg_l` is also constructed using the factory interface.

4.2.8 Additional Aspects of the Example

So far we have covered most of the example from Appendix A.1. Also covered in this example is how detailed error information can be queried through the API and how to get additional information on the status of individual modules. We will not go into further details with the example here, but refer to the interfaces `VDMErrors` and `VDMModuleRepos` of the IDL description as well as the example source code and comments of Appendix A.1 for more information.

4.3 Compiling the Client

To successfully compile the file `client_example.cc` each of the following requirements must be fulfilled:

- The omniORB must have been successfully installed If the binary distribution is not available for your particular platform it must be compiled as well. Moreover, the `PATH` environment variable must point to the binaries directory of omniORB.
- The following files, found in `$TOOLBOX/api/corba`, must be present (where `$TOOLBOX` represents the directory in which the Toolbox was installed).
  - `client_example.cc`
  - `corba_client.h`, `corba_client.cc`
  - `corba_api.idl`, `metaiv_idl.idl`
  - `Makefile`, `Makefile.nm`
- Your VDM Toolbox must contain the VDM C++ library, i.e. the include file `metaiv.h` and the library `libvdm.a` (Unix) or `vdm.lib` (Windows 2000/XP/Vista).

To compile the example you can simply use the makefile. On Linux you run `make` with `Makefile`, while on Windows 2000/XP/Vista you use `nmake` with

Makefile.nm. You must modify the macros OMNIDIR and TBDIR of the make file to point to the installation directory of omniORB and the VDM Toolbox respectively.

Note that if you wish to use Microsoft's Foundation Classes under win32, the MFC library should be statically linked.

4.3.1 Supported Compilers

The client example in Appendix A.1 has been compiled and tested on Microsoft Windows 2000/XP/Vista and Microsoft Visual C++ 2005 SP1, and on Linux with GNU gcc 3, 4.

4.4 Running the Client

Before you run the client example you must ensure that a VDM Toolbox to be used as server is currently running. Use the VDM_OBJECT_LOCATION environment variable in order to tell the client where to look for the [vdm|vpp]ref.ior file.

5 Writing a Java Client

5.1 Choosing a CORBA Implementation

The Java 1.3 API contains a package called `org.omg.CORBA`, that provides the mapping of the OMG CORBA APIs to the Java programming language. The package includes the class `ORB`, which is implemented so that a programmer can use it as a fully-functional Object Request Broker.

The example in the following will use this CORBA implementation.

In addition to a CORBA implementation, the user needs to have access to the described IDL modules: `corba_api.idl` and `metaiv_idl.idl`. These have been translated to Java packages and classes and can be used by including the `ToolboxAPI.jar` file in the classpath. This file is part of the Toolbox distribution, in the `api/corba` sub-directory.

It contains three packages:

- `jp.co.csk.vdm.toolbox.api.corba.VDM`

  This package contains the VDM module defined in `metaiv_idl.idl`. It contains consequently a Java interface for every VDM value.

- `jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI`

  This package contains the interfaces from `corba_api.idl`.

- `jp.co.csk.vdm.toolbox.api`

  This package contains only one class, called `ToolboxClient`. It implements methods used to connect client applications to the VDM Toolbox through the VDM Toolbox CORBA API.

All three packages are documented by HTML documentation generated by the *javadoc* program. Both the `ToolboxAPI.jar` file and the HTML documentation are distributed with the VDM Toolbox.

If you don't use the CORBA implementation following with Java 1.3, you have to translate the IDL files to Java yourself. The files in `ToolboxAPI.jar` have been created using the `idltojava` compiler (downloadable from the Java Developer Connection): `http://developer.java.sun.com`. If you are using the Sun JDK 1.3 an executable `idlj` will be part of the distribution. It is the SUN IDL to Java compiler, that generates the Java stubs and skeletons for you.

5.2 Implementing a Client

In this Section we go through an example illustrating how to use the VDM Toolbox from a client written in Java. In the following presentation we will show excerpts from a complete example, which can be seen in full length in Appendix A.2.

5.2.1 Importing CORBA Services

Your client program should start by importing the `org.omg.CORBA` package together with the three packages described above:

```
import org.omg.CORBA.*;

import jp.co.csk.vdm.toolbox.api.ToolboxClient;
import jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.*;
import jp.co.csk.vdm.toolbox.api.corba.VDM.*;
```

5.2.2 Acquiring the Application Object

As in the C++ implementation, the approach used to acquire the main handle to the VDM Toolbox - the `VDMApplication` CORBA object - is to let the client either resolve the reference from the COS NamingService or to read a stringified object reference (created by the most recently started VDM Toolbox) and convert this to a CORBA object reference.

The easiest possibility to get hold of a CORBA-reference to the `VDMApplication` CORBA object is to use the `getVDMApplication` method that you can find in the above mentioned `ToolboxClient.java` file.

When you start either the VDM-SL or the VDM++ Toolbox, it will check if a COS NameService is running. The ORB will search for a configuration file. You can specify the location of this file using the `OMNIORB_CONFIG` environment variable (refer to the omniORB-documentation to see how to use the registry for this if you use Windows). A typical `omniORB.cfg` file will contain following entries:

```
ORBInitialHost gandalf
ORBInitialPort 2809
```

This means, that the NamingService is running on a host called `gandalf` at port 2809. omniORB provides such a NameService (there should be an executable called `omniNames` that is part of the omniORB-distribution), but you can use virtually any other CORBA-compliant NameService as long as you make it known to the omniORB using the `omniORB.cfg` file. Please refer to the omniORB documentation for further details. The Toolbox has been tested with the `tnameserv`-NamingService from the Sun JDK1.3, too. You will have to tell your client application where it can find the NameService. You can either do this by using the command-line-parameters `-ORBInitialPort <port> -ORBInitialHost <host>`, or directly in the source code, by setting the corresponding properties.

```
Properties props = new Properties ();
props.put ("org.omg.CORBA.ORBInitialHost", "gandalf");
props.put ("org.omg.CORBA.ORBInitialPort", 2809);
orb = ORB.init (args, props);
```

The VDM-SL Toolbox binds its `VDMApplication` object to the name `SL_TOOLBOX`, of kind `VDMApplication`, while Application object of the VDM++ Toolbox uses the name `PP_TOOLBOX`, of kind `VDMApplication`. This makes it possible for the client to distinguish the objects, so that it is no problem to run an instance of each Toolbox at the same time.

The following code is used to resolve the `VDMApplication`-object from the NameService:

```
org.omg.CORBA.Object obj =
   orb.resolve_initial_references ("NameService");
NamingContext ctx = NamingContextHelper.narrow (obj);

NameComponent nc = null;

if (toolType == ToolType.SL_TOOLBOX)
   nc = new NameComponent ("SL_TOOLBOX", "VDMApplication");
else
   nc = new NameComponent ("PP_TOOLBOX", "VDMApplication");

NameComponent[] name = {nc};

org.omg.CORBA.Object obj = // use full qualified classpath!
   ctx.resolve (name);
VDMApplication app = VDMApplicationHelper.narrow (obj);
```

Take care that you do not run two instances of the same Toolbox, because then only the `VDMApplication` object of the Toolbox that has been started first will be accessible for the client. It is no problem to run more than one client using the same `VDMApplication` object, but keep in mind that they will influence each other.

If the `getVDMApplication` method cannot locate a NamingService, it will try to resolve the `VDMApplication`-reference by using the string reference file. All ORB implementations must implement the two functions `object_to_string` and `string_to_object`, used to encode and decode object references. The VDM Toolbox uses `object_to_string` to encode the application object as a string and writes this string to a file. Subsequently the client must read this file and use `string_to_object` to convert the string to an object reference. The file generated by the VDM Toolbox is named `vdmref.ior` or `vppref.ior`, and it is written in the location specified by the `VDM_OBJECT_LOCATION` environment variable. If the environment variable is not set, the file is located in the root of your home directory (as pointed to by `$HOME`) if the VDM Toolbox is running on Unix, and in your profile directory (as pointed to by `%USERPROFILE%`) if running on Windows 2000/XP/Vista.

The method `readRefFile` of class `ToolboxClient` is used by `getVDMApplication` to read this `vdmref.ior` or `vppref.ior` file created by the Toolbox. You can establish the connection to the Toolbox denoted by the object reference string by calling the `getVDMApplication` method of the `Toolbox` client class. This method returns an object reference to the CORBA `VDMApplication` object.

```
VDMApplication app = ToolboxClient.getVDMApplication(args,ref);
```

After the call to `getVDMApplication` the variable `app` is the main handle to the VDM Toolbox.

The client has to register itself in the server before performing any calls towards the server. Similarly, it has to unregister itself when it terminates. This is done by calling the `Register` and `Unregister` methods of the `VDMApplication` class.

```
short client_id = app.Register();
...
...
app.Unregister(client_id);
```

We are now in the position to access services of the running VDM Toolbox.

5.2.3 Configuring the Current Project

The following lines of code acquire a handle to the VDMProject interface of the VDM Toolbox and use the New and AddFile methods of this interface. As a result the project of the VDM Toolbox is configured to contain a single file, sort.vdm. Files added to the project in this way must be located in the same directory as where the VDM Toolbox was started. Otherwise the file name must be given with its absolute path. If the client tries to add a non-existing file the server will throw an exception of type APIError indicating the error. Exceptions are described in Section 3.3.

```
VDMProject prj = app.GetProject();
prj.New();
prj.AddFile("sort.vdm");
```

5.2.4 Using the Parser

To use the parser from a client you must get a handle to the VDMParser interface, and to parse a file you call the Parse method of this interface with the file name as its single argument. I.e.,

```
VDMParser parser = app.GetParser();
parser.Parse("sort.vdm");
```

will parse the file sort.vdm.

Alternatively you could use the VDMProject interface to get the list of files configured for the current project and then parse each file of this list:

```
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;

for(int i=0; i<flist.length; i++){
   System.out.println("...Parsing" + flist[i] + "...");
   if(parser.Parse(flist[i]))
        System.out.println("done.");
   else
```

```
        System.out.println("error.");
}
```

This example illustrates several important aspects of the API. Initially we declare `fl` to be a list of files and use `GetFiles` to retrieve the list of files in the current project. From the IDL description of the API you will see that the type `FileList` is defined as an unbounded sequence of strings. Consequently the list of files, `fl`, has all methods of the IDL sequence as stated by the CORBA specification [OMG96]. The length of a IDL sequence can be accessed through the method `length`, and the individual elements can be indexed as ordinary arrays in Java.

Moreover, this example shows, that the support for out and inout parameter passing modes in Java requires the use of additional "holder" classes. These classes are available for all of the basic IDL datatypes in the `org.omg.CORBA` package and are generated for all named user defined types except those defined by typedefs.

For user defined IDL types, the holder class name is constructed by appending `Holder` to the mapped (Java) name of the type. Each holder class has a public instance member, `value`, which is the typed value.

To summarize: The above lines of code retrieve the list of files in the current project, iterate through the list, and for each item calls the `Parse` method to parse each file. Notice that `Parse` returns a boolean value indicating the success of parsing the file. Parsing all files of the project by iterating the list of files is actually more complicated than need to be. Instead you could use the `ParseList` method:

```
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;
parser.ParseList(flist);
```

5.2.5 Using the Type Checker

The interface of the type checker is similar to the interface of the parser. The interface has a number of attributes that can be accessed and modified by the client. Attributes can be read and modified for example:

```
// Get the value of DefTypeCheck:
```

```
boolean dtc = tpck.DefTypeCheck();

// Set the value of ExtendedTypeCheck to true
tpck.ExtendedTypeCheck(true);
```

provided of course that `tpck` is a valid handle to the type checker interface.

5.2.6 Using the Interpreter

The following example illustrates how `EvalExpression` of the interpreter interface can be used to have the interpreter evaluate any VDM expression.

```
VDMInterpreter interp = app.GetInterpreter();
Generic g;
g = interp.EvalExpression(client_id,
        "[e|e in set {1,...,20} & \
              exists1 x in set {2,...,e} & e mod x = 0]");

if(g.IsSequence()){
     System.out.println("All primes below 20": " + g.ascii());
}
```

The string passed to `EvalExpression` is evaluated and the result of the evaluation is returned as a VDM value in a `Generic`, which can later be used in a call to `Apply`, or read/modified by the methods provided by the interface of the VDM values as described in Section 3.2.

The following example illustrates how to use a function of the VDM specification:

```
interp.Init();
g = interp.EvalExpression(client_id, "MergeSort([6,4,9,7,3,42])");
```

Notice that before you can call any function of the specification you must make sure that the interpreter is initialized.

An alternative to `EvalExpression` is to use the method `Apply`, which takes as argument the name of the function or operation to apply and a sequence of arguments for the function or method. The following example creates a VDM sequence of integers to be sorted with `MergeSort`:

```
VDMFactory fact = app.GetVDMFactory();
Sequence list = fact.MkSequence(client_id);
Numeric elem;
for(int j=0; j<20; j++){
    elem = fact.MkNumeric(client_id, j);
    list.ImpPrepend(elem);
}
```

The resulting sequence, `list`, now contains the integers from 19 down to 0. Notice how VDM values are constructed in the client by using the `VDMFactory` interface.

To call `MergeSort` through `Apply` we have to construct the list of arguments. The arguments for the function to be called through `Apply` are contained in a `Sequence`. The function we want to call only takes one argument, the sequence of integers we have just constructed:

```
Sequence arg_l = fact.MkSequence(client_id);
arg_l.ImpAppend(list);
```

Now `MergeSort` can be applied as follows:

```
g = interp->Apply(client_id, "MergeSort", arg_l);
```

if, of course, the interpreter has been initialized. Notice that the argument list `arg_l` is also constructed using the factory interface.

Finally we show how to iterate through the returned sequence to compute the sum of all the elements of the sequence:

```
Sequence s = SequenceHelper.narrow(g);
GenericHolder eholder = new GenericHolder();
int sum=0;
for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
  Numeric num = NumericHelper.narrow(eholder.value);
  sum = sum + (int) num.GetValue();
}
```

5.2.7 Additional Aspects of the Example

So far we have covered most of the example from Appendix A.2. Also covered in this example is how detailed error information can be queried through the API and how to get additional information on the status of individual modules. Moreover, it shows how to convert distributed VDM values to "real" VDM Java values. We will not go into further details with the example here, but refer to the interfaces `VDMErrors` and `VDMModuleRepos` of the IDL description as well as the example source code and comments of Appendix A.2 for more information.

5.3 Compiling the Client

The `client_example.java` file must be compiled using the following compiler:

```
jdk1.3
```

You can compile the main program by writing:

```
javac client_example.java
```

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

```
CLASSPATH=ToolboxAPI_Library/ToolboxAPI.jar:$CLASSPATH
export CLASSPATH
```

Replace `ToolboxAPI_Library` with the name of the directory in which the file `ToolboxAPI.jar` is installed.

If you are working on a Windows-based system, you can use the following command within the Windows command interpreter:

```
set CLASSPATH=ToolboxAPI_Library/ToolboxAPI.jar;$CLASSPATH
```

Note that for Windows you must use "`;`" and not "`:`" as the delimiter.

5.4 Running the Client

Before you run the client example you must first ensure that a VDM Toolbox to be used as server is currently running. In order to make the example work, you need a CORBA enabled Toolbox.

By default the example program assumes that it is running with a VDM-SL Toolbox running on Linux. In this case simply run

```
java client_example.java
```

To run on windows you must set the WIN property and to run with a VDM++ Toolbox you must set the VDMPP property.

```
java -DVDMPP -DWIN client_example
```

6 API Reference Guide

6.1 Corba API

6.1.1 Types

The following type synonyms are defined:

Name	Synonym for
`ModuleName`	`string`
`ModuleList`	`sequence<ModuleName>`
`ClassName`	`string`
`ClassList`	`sequence<ClassName>`
`FileName`	`string`
`FileList`	`sequence<FileName>`
`ErrorList`	`sequence<Error>`

The following enumeration is defined

```
enum ToolType {SL_TOOLBOX, PP_TOOLBOX};
```

The following structures are defined:

6.1.2 Error Structure

Field	Meaning
`FileName fname`	Name of file in which error/warning was found.
`unsigned short line`	Line number of error/warning.
`unsigned short col`	Column number of error/warning.
`string msg`	Text of error/warning.

6.1.3 ModuleStatus Structure

Field	**Meaning**
`boolean SyntaxChecked`	Attribute describing whether module (or class) has been syntax checked.
`boolean TypeChecked`	Attribute describing whether module (or class) has been type checked.
`boolean CodeGenerated`	Attribute describing whether module (or class) has been code generated.
`boolean PrettyPrinted`	Attribute describing whether module (or class) has been pretty printed.

6.1.4 VDMApplication Interface

Name	**Description**
`readonly attribute ToolType Tool`	Returns the type of tool for server Toolbox.
`ClientID Register()`	Returns a unique client id. A client should register itself with the server before performing any API calls.
`void Unregister(ClientID id)`	Releases any resources associated with client `id`. This should be called when a client terminates.
`VDMProject GetProject()`	Returns a handle to the current project.
`VDMInterpreter GetInterpreter()`	Returns a handle to the Toolbox Interpreter.
`VDMCodeGenerator GetCodeGenerator()`	Returns a handle to the Toolbox Code Generator.
`VDMParser GetParser()`	Returns a handle to the Toolbox Parser.

Name	Description
VDMTypeChecker GetTypeChecker()	Returns a handle to the Toolbox Type Checker.
VDMPrettyPrinter GetPrettyPrinter()	Returns a handle to the Toolbox Pretty Printer.
VDMErrors GetErrorHandler()	Returns an handle to the Toolbox errors interface.
VDMModuleRepos GetModuleRepos()	Returns a handle to the Toolbox module (or class) repository.
VDMFactory GetVDMFactory()	Returns a handle to a VDM value factory. Since CORBA 2.x does not support remote object instantiation, this factory must be used to create CORBA VDM Objects (e.g. VDMSequence, VDMToken. . .)
void PushTag(in ClientID id)	Generates unique tag for client id and pushes this onto the Toolbox's internal tag stack. All objects created by the Toolbox for this client thereafter are tagged with this tag.
void DestroyTag(in ClientID id) raises APIError	Pops the topmost tag on the tag stack for ClientID and destroys all objects tagged with this value.

6.1.5 VDMCodeGenerator Interface

Name	Description
attribute boolean GeneratePosInfo	Enables or disables generation of position information. This allows all constructs in the generated code to be traced back to the specification. Default is false.
enum LanguageType CPP, JAVA	Possible target languages of the code generator

Name	Description
`boolean GenerateCode(in ModuleName name, in LanguageType targetLang) raises APIError`	Generates C++/Java code (depending on the targetLang flag) for module (or class) `name`. Raises an exception if `name` is not a valid module or class name in the current project or if you try to generate Java code for a VDM-SL module (since Java code generation is only available for VDM++-classes).
`boolean GenerateCodeList(in ModuleList names) raises APIError`	Generates C++/Java code for each module (or class) named in `names`. Raises an exception if any name in `names` is not a valid module or class name in the current project or if you try to generate Java code for a VDM-SL module.

6.1.6 VDMErrors Interface

Name	Description
`readonly attribute unsigned short NumErr`	Returns the number of errors generated by the most recent action.
`readonly attribute unsigned short NumWarn()`	Returns the number of warnings generated by the most recent action.
`unsigned short GetErrors(out ErrorList err)`	Returns the list of errors generated by the most recent action in `err`.
`unsigned short GetWarnings(out ErrorList err)`	Returns the list of warnings generated by the most recent action in `err`.

6.1.7 VDMInterpreter Interface

Name	Description
`attribute boolean DynTypeCheck`	Enables or disables dynamic type checking. Default is false.
`attribute boolean DynInvCheck`	Enables or disables dynamic invariant checking. Default is false. If true, the `DynTypeCheck` attribute is automatically set to true.
`attribute boolean DynPreCheck`	Enables or disables dynamic precondition checking. Default is false.
`attribute boolean DynPostCheck`	Enables or disables dynamic postcondition checking. Default is false.
`attribute boolean PPOfValues`	Enables or disables pretty printing of values. Default is true.
`attribute boolean Verbose`	Enables or disables verbose interaction with the Toolbox i.e. whether the result of actions performed via the API are echoed in the interpreter window. Default is false.
`attribute boolean Debug`	Enables or disables debug mode. In this mode breakpoints in the specification are respected. When a breakpoint is reached evaluation is suspended and the user must interact with the graphical user interface to do the actual debugging. Default is false.
`void Initialize()`	Initializes the interpreter. Must be done before evaluation.
`VDMGeneric EvalExpression(in ClientID id, in string expr) raises APIError`	Evaluates `expr` on behalf of client with id `id`. Result of evaluation returned as result of method. Result will be echoed to screen if `Verbose` is true. Run-time errors cause exceptions to be raised.

Name	Description
`VDMGeneric Apply(in ClientID id, in string f, in VDMSequence arg) raises APIError`	Applies the function (or operation) `f` on argument(s) `arg` on behalf of client `id`. The result of function (or operation) call is returned as result of method. Run-time errors cause exceptions to be raised.
`void EvalCmd(in string cmd)`	Evaluates the command `cmd` as if it was written directly to the interpreter.
`long SetBreakPointByPos(in string file, in long line, in long col)`	Sets a breakpoint at the specified position (line, column) in the specified file and returns the number of the new breakpoint. An exception or a return value of -1 indicates, that an error has occured (e.g. the file does not exists or the specified line number is not valid).
`long SetBreakPointByName(in string mod, in string func) raises APIError`	Sets a breakpoint at the specified function (func) in the specified module and returns the number of the new breakpoint. An exception or a return value of -1 indicates, that an error has occured (e.g. the module or the function does not exist).
`void DeleteBreakPoint(in long num) raises APIError`	Used to delete a breakpoint. It takes the number returned by the breakpoint setting methods as a parameter.

Name	Description
VDMTuple StartDebugging (in ClientID id, in string expr) raises APIError	Starts debugging an expression. This method returns, if the evaluation of the expression has been finished or if an breakpoint has been encountered. It returns a VDMTuple containing the evaluation state (which can be either <BREAKPOINT>, <INTERRUPT>, <SUCCESS> or <ERROR>) and, in case of <SUCCESS> (what means, that the expression has been successfully evaluated) the MetaIV value that represents the evaluation result.
VDMTuple DebugStep (in ClientID id) raises APIError	This method is the equivalent to the step command in the toolbox. It executes the next statement and then breaks. It will not step into function and operation calls. It returns a VDMTuple containing the evaluation state (which can be either <BREAKPOINT>, <INTERRUPT>, <SUCCESS> or <ERROR>) and, in case of <SUCCESS> (what means, that the expression has been successfully evaluated) the result of the evaluation as a MetaIV value.
VDMTuple DebugStepIn (in ClientID id) raises APIError	This method is the equivalent to the stepin command in the toolbox. It executes the next statement and then breaks. It will also step into function and operation calls. It returns a VDMTuple containing the evaluation state (which can be either <BREAKPOINT>, <INTERRUPT>, <SUCCESS> or <ERROR>) and, in case of <SUCCESS> (what means, that the expression has been successfully evaluated) the result of the evaluation as a MetaIV value.

Name	Description
`VDMTuple DebugSingleStep (in ClientID id) raises APIError`	This method is the equivalent to the `singlestep` command in the toolbox. It executes the next expression or statement and then breaks. It returns a `VDMTuple` containing the evaluation state (which can be either `<BREAKPOINT>`, `<INTERRUPT>`, `<SUCCESS>` or `<ERROR>`) and, in case of `<SUCCESS>` (what means, that the expression has been successfully evaluated) the result of the evaluation as a MetaIV value.
`VDMTuple DebugContinue (in ClientID id) raises APIError`	This method is the equivalent to the `cont` command in the toolbox. It continues the execution after a breakpoint has been encountered. It returns a `VDMTuple` containing the evaluation state (which can be either `<BREAKPOINT>`, `<INTERRUPT>`, `<SUCCESS>` or `<ERROR>`) and, in case of `<SUCCESS>` (what means, that the expression has been successfully evaluated) the result of the evaluation as a MetaIV value.

6.1.8 VDMModuleRepos Interface

Name	Description
`unsigned short FilesOfModule(out FileList files, in ModuleName name)`	Delivers names of the files containing module (or class) `name` in `files`. The sequence will consist of exactly one name unless this is a flat module in which case the module name `DefaultMod` may be spread across several files. Returns the number of files.

Name	Description
void Status(out ModuleStatus state, in ModuleName name) raises APIError	Delivers in state the status of module (or class) name. Raises an exception if name does not exist in the current project.
unsigned short SuperClasses(out ClassList classes, in ClassName name) raises APIError	Delivers in classes list of superclasses of class name. VDM++ specific, raises an exception if called with the VDM-SL toolbox.
unsigned short SubClasses(out ClassList classes, in ClassName name) raises APIError	Delivers in classes list of subclasses of class name. VDM++ specific, raises an exception if called with the VDM-SL toolbox.
unsigned short UsesOf(out ClassList classes, in ClassName name) raises APIError	Delivers in classes list of classes used by class name. VDM++ specific, raises an exception if called with the VDM-SL toolbox.
unsigned short UsedBy(out ClassList classes, in ClassName name) raises APIError	Delivers in classes list of classes that use class name. VDM++ specific, raises an exception if called with the VDM-SL toolbox.

6.1.9 VDMParser Interface

Name	Description
boolean Parse(in FileName name) raises APIError	Returns true if file name was successfully parsed; otherwise returns false and the state of the VDMErrors interface is modified. Raises an exception if the file does not exist.
boolean ParseList(in FileList names) raises APIError	Returns true if all of the files in names were successfully parsed. Otherwise returns false and the state of the VDMErrors interface is modified. Raises an exception if any file does not exist.

6.1.10 VDMPrettyPrinter Interface

Name	Description
`boolean PrettyPrint(in FileName name) raises APIError`	Returns `true` if module (or class) `name` was successfully pretty printed; otherwise returns `false` and the state of the `VDMErrors` interface is modified. Raises an exception if the module (or class) does not exist.
`boolean PrettyPrintList(in FileList names) raises APIError`	Returns `true` if all of the modules (or classes) in `names` were successfully pretty printed. Otherwise returns `false` and the state of the `VDMErrors` interface is modified. Raises an exception if any module (or class) does not exist.

6.1.11 VDMProject Interface

Name	Description
`void New()`	Creates a new project
`void Open(in FileName name) raises APIError`	Opens project with name given by argument `FileName`.
`void Save() raises APIError`	Save project using existing name. Raises an exception if the project currently has no name (e.g. if it is new)
`void SaveAs(in FileName name)`	Save project using name given by argument `FileName`
`unsigned short GetModules(out ModuleList modules)`	For current project, generates list of modules (VDM-SL) or list of classes (VDM++) in `modules` and returns the number of modules or classes.
`unsigned short GetFiles(out FileList files)`	For current project, generates list of files in `files` and returns the number of files.

Name	Description
`void AddFile(in FileName name) raises APIError`	Adds a file to the project. Raises `APIError` if unsuccessful (e.g. file not found).
`void RemoveFile(in FileName name) raises APIError`	Removes a file from the project. Raises `APIError` if unsuccessful (e.g. file not in current project) .

6.1.12 VDMTypeChecker Interface

Name	Description
`attribute boolean DefTypeCheck`	Determines whether type checking mode is "def" (true) or "pos" (false). Default is "pos".
`attribute boolean ExtendedTypeCheck`	Determines whether extended type checking is enabled. Default is false.
`boolean TypeCheck(in ModuleName name) raises APIError`	Returns true if module (or class) `name` was successfully type checked; otherwise returns false and the state of the `VDMErrors` interface is modified. Raises an exception if the module (or class) does not exist.
`boolean TypeCheckList(in ModuleList names) raises APIError`	Returns true if all of the modules (or classes) in `names` were successfully type checked. Otherwise returns false and the state of the `VDMErrors` interface is modified. Raises an exception if any module (or class) does not exist.

6.2 VDM API

In the following, Section titles have fully qualified interface names (i.e. VDM::Interface) but short names used in actual descriptions for brevity.

6.2.1 Types

The following type synonyms are defined:

Name	Synonym for
`ClientID`	`short`
`bytes`	`sequence<octet>`

6.2.2 VDM::VDMGeneric Interface

Name	Description
`string ToAscii()`	Returns a string representation of the object.
`boolean IsNil()`	Returns true if and only if the object has type `VDMNil`.
`boolean IsChar()`	Returns true if and only if the object has type `VDMChar`
`boolean IsNumeric()`	Returns true if and only if the object has type `VDMNumeric`
`boolean IsQuote()`	Returns true if and only if the object has type `VDMQuote`
`boolean IsTuple()`	Returns true if and only if the object has type `Tuple`
`boolean IsRecord()`	Returns true if and only if the object has type `Record`
`boolean IsSet()`	Returns true if and only if the object has type `Set`
`boolean IsMap()`	Returns true if and only if the object has type `Map`
`boolean IsText()`	Returns true if and only if the object has type `VDMText`
`boolean IsToken()`	Returns true if and only if the object has type `VDMToken`
`boolean IsBool()`	Returns true if and only if the object has type `VDMBool`
`boolean IsSequence()`	Returns true if and only if the object has type `Sequence`

Name	Description
boolean IsObjectRef()	Returns true if and only if the object is a reference to another VDM object.
void Destroy() raises APIError	Calls to this method indicate to the server that the client has no further use for this object. If this was the last reference to the server object, the resources associated with it will be released.
bytes GetCPPValue()	Returns the binary representation of the MetaIV value. In this way, by linking the client application with the VDM library, it is possible to create a 'real' MetaIV value in the client. This allows for more efficient access when iterating through a large VDM value.
VDMGeneric Clone()	This method returns a copy of the value held by the object on which this method is invoked.

6.2.3 Basic VDM Types

The following interfaces extend the VDMGeneric interface. The only difference is the addition of a GetValue() method which has default access and returns the value corresponding to this VDM value.

Interface	GetValue() returns
VDM::VDMBool	boolean
VDM::VDMChar	char
VDM::VDMNumeric	double
VDM::VDMQuote	string
VDM::VDMText	string
VDM::VDMToken	string

The interface VDM::VDMNil has no public methods or member variables in addition to those it inherits.

6.2.4 VDM::VDMMap Interface

This interface extends VDMGeneric.

Name	Description
`void Insert(in VDMGeneric key, in VDMGeneric val) raises VDMError`	Adds a new key `key` with value `val` to the map. Raises an exception if `key` is already in the domain of the map.
`void ImpModify(in VDMGeneric key, in VDMGeneric val)`	Modifies the map so that `key` has value `val`.
`VDMGeneric Apply(in VDMGeneric key) raises VDMError`	Returns the value corresponding to `key`. Raises an exception if `key` is not in the domain of the map.
`void ImpOverride(in VDMMap m)`	Overrides this map with the map object `m`.
`unsigned long Size()`	Returns the number of keys in the map.
`boolean IsEmpty()`	Returns true if and only if the map has no keys.
`Set Dom()`	Returns the domain (keys) of the map.
`Set Rng()`	Returns the range (values) of the map.
`boolean DomExists(in VDMGeneric g)`	Returns true if and only if `g` lies in the domain of the map.
`void RemElem(in VDMGeneric key) raises VDMError`	Removes key `key` from the map. Raises an exception if `key` is not in the domain of the map.
`short First(out VDMGeneric g)`	Delivers the first key in the map in `g`. Returns 1 if the map is non-empty, 0 if the map is empty.
`short Next(out VDMGeneric g)`	Iterator delivering the next key in the map in `g`. Returns 1 if there are still keys in the map not yet visited, 0 if all of the keys have been yielded by the iterator.

6.2.5 VDM::VDMRecord Interface

This interface extends Generic.

Name	Description
`void SetField(in unsigned long i, in VDMGeneric g) raises VDMError`	Sets field `i` to have value `g`. Raises an exception if `i` is not a valid field for this record (i.e. not in the range $1, \ldots, number\ of\ fields$)
`VDMGeneric GetField(in unsigned long i) raises VDMError`	Returns the value of field `i`. Raises an exception if `i` is not a valid field for this record (i.e. not in the range $1, \ldots, number\ of\ fields$)
`string GetTag()`	Returns the tag of this record.
`boolean Is(in string tag)`	Returns true if and only if `tag` matches the tag for this record.
`long Length()`	Returns the number of fields in this record.

6.2.6 VDM::VDMSequence Interface

This interface extends VDMGeneric.

Name	Description
`VDMGeneric Index(in long i) raises VDMError`	Returns the value at index `i` in the sequence. Raises an exception if `i` is not a valid index.
`VDMGeneric Hd() raises VDMError`	Returns the value at the head of the sequence. Raises an exception if the sequence is empty.
`VDMSequence Tl() raises VDMError`	Returns the tail of the sequence, leaving this sequence unchanged. Raises an exception if the sequence is empty.
`void ImpTl() raises VDMError`	Removes the head of this sequence. Raises an exception if the sequence is empty.
`void RemElem(in long i) raises VDMError`	Removes the element at index `i` from the sequence.
`long Length()`	Returns the length of the sequence

Name	Description
`boolean GetString(out string s)`	If this sequence is purely a sequence of char, returns true, and delivers the corresponding string in s, otherwise returns false.
`boolean IsEmpty()`	Returns true if and only if the sequence is empty.
`void ImpAppend(in VDMGeneric g)`	Appends value `g` to the end of the sequence.
`void ImpModify(in long i, in VDMGeneric g) raises VDMError`	Overwrites the value stored at index `i` with `g`. Raises an exception if `i` is not a valid index for this sequence (i.e. not in the range $1, \ldots, \mathit{length\ of\ sequence}$)
`void ImpPrepend(in VDMGeneric g)`	Prepends the value `g` to the front of the sequence.
`void ImpConc(in VDMSequence s)`	Concatenates sequence `s` to the end of this sequence.
`Set Elems()`	Returns the set of elements in the sequence.
`short First(out VDMGeneric g)`	Delivers the first element in the sequence in `g`. Returns 1 if the sequence is non-empty, 0 if the sequence is empty.
`short Next(out VDMGeneric g)`	Iterator delivering the next element in the sequence in `g`. Returns 1 if there are still elements in the sequence not yet visited, 0 if all of the elements have been yielded by the iterator.

6.2.7 VDM::VDMSet Interface

This interface extends VDMGeneric.

Name	Description
`void Insert(in VDMGeneric g)`	Inserts value `g` into the set.
`unsigned long Card()`	Returns the cardinality of the set.
`boolean IsEmpty()`	Returns true if and only if the set is empty.

Name	Description
`boolean InSet(in VDMGeneric g)`	Returns true if and only if `g` lies in the set.
`void ImpUnion(in VDMSet s)`	Adds all of the elements of `s` to this set.
`void ImpIntersect(in VDMSet s)`	Removes from this set those elements that do not occur in `s`.
`VDMGeneric GetElem() raises VDMError`	Returns an arbitrary element of the set. Raises an exception if the set is empty.
`void RemElem(in VDMGeneric g) raises VDMError`	Removes the element `g` from the set. Raises an exception if `g` does not occur in the set.
`boolean SubSet(in VDMSet s)`	Returns true if and only if `s` is a subset of this set.
`void ImpDiff(in VDMSet s)`	Modifies this set by removing any elements from it that also occur in the set `s`.
`short First(out VDMGeneric g)`	Delivers the first element in the set in `g`. Returns 1 if the set is non-empty, 0 if the set is empty.
`short Next(out VDMGeneric g)`	Iterator delivering the next element in the set in `g`. Returns 1 if there are still elements in the set not yet visited, 0 if all of the elements have been yielded by the iterator.

6.2.8 VDMTuple Interface

This interface extends VDMGeneric.

Name	Description
`void SetField(in unsigned long i, in VDMGeneric g) raises VDMError`	Sets field `i` in the tuple to be value `g`. Raises an exception if field `i` does not exist.
`VDMGeneric GetField(in unsigned long i) raises VDMError`	Returns field `i`. Raises an exception if field `i` does not exist.
`unsigned long Length()`	Returns the number of fields in the tuple.

6.2.9 VDMFactory Interface

Name	Description
`VDMNumeric MkNumeric(in ClientID id, in double d)`	Returns a VDMNumeric object with value `d` to client `id`
`VDMBool MkBool(in ClientID id, in boolean b);`	Returns a VDMBool object with value `b` to client `id`
`VDMNil MkNil(in ClientID id);`	Returns a VDMNil object to client `id`
`VDMQuote MkQuote(in ClientID id, in string s);`	Returns a VDMQuote object with value `s` to client `id`
`VDMChar MkChar(in ClientID id, in char c);`	Returns a VDMChar object with value `c` to client `id`
`VDMText MkText(in ClientID id, in string s);`	Returns a VDMText object with value `s` to client `id`
`VDMToken MkToken(in ClientID id, in string s);`	Returns a VDMToken object with value `s` to client `id`
`VDMMap MkMap(in ClientID id);`	Returns a VDMMap object to client `id`
`VDMSequence MkSequence(in ClientID id);`	Returns a VDMSequence object to client `id`
`VDMSet MkSet(in ClientID id)`	Returns a VDMSet object to client `id`
`VDMTuple MkTuple(in ClientID id, in unsigned long length);`	Returns a VDMTuple object with `length` components to client `id`
`VDMGeneric FromCPPValue(in ClientID id, in bytes cppvalue)`	Converts a 'real' MetaIV value, in its binary representation, to a VDMGeneric. This function is the 'inverse' of GetCPPValue().

6.3 Exceptions

Two exceptions are defined:

Exception	Component
`ToolboxAPI::APIError`	`string msg`
`VDM::VDMError`	`short err`

The value returned in a `VDMError` exception packet is a status code. A list of the possible status codes, and their meaning is given below.

Value	Description
1	Attempt to insert key into map which already exists with different range value
2	Not in domain
4	Index out of range
6	Op on empty set
7	Not in set
10	Head on empty sequence
11	Tail on empty sequence
12	Range error

6.4 C++ API Reference

In this Section we briefly describe the translation of the IDL interfaces described in Sections 6.1 and 6.2. This is based on the translation generated by the omniORB IDL Compiler (Version 2.6.1).

6.4.1 corba_client.h

A file `corba_client.cc` is provided to simplify initialization of the ORB. Its interface is defined in `corba_client.h` and is listed here.

The enumerated type `GetAppReturnCode` is declared with values listed in the following table:

Value	Description
`VDM_OBJECT_LOCATION_NOT_SET`	The environment variable `VDM_OBJECT_LOCATION` was not set. See the function `GetAppReturnCode` for details.

`OBJECT_STRING_NON_EXISTING`	The VDM Toolbox was not running
`CORBA_SUCCESS`	Successful communication with the VDM Toolbox
`CORBA_ERROR`	Error in communication with VDM Toolbox

The functions defined are as follows:

Name	**Description**
`void init_corba(int argc, char *argv[])`	Initializes the CORBA ORB and BOA (Basic Object Adapter). Call this function before using any other CORBA related functions. For more information on the Object Request Broker and the Basic Object Adapter refer to the CORBA specification, which is available from the OMB CORBA homepage (http://www.corba.org).
`GetAppReturnCode get_app(VDMApplication_var app, char *path [, ToolboxAPI::ToolType toolType])`	Tries to resolve VDMApplication from the CosNamingService. If no NamingService is running, it reads a file named '`vdmref.ior`' or '`vppref.ior`' to get the id of the running server. The file must be located in the directory pointed to by the environment variable `VDM_OBJECT_LOCATION` (unless you provide a path). The `ToolType` (either `ToolboxAPI::SL_TOOLBOX` or `ToolboxAPI::PP_TOOLBOX` is optional, `SL_TOOLBOX` is the default setting. The value returns indicates the result of the operation.
`Generic GetCPPValue(VDM::Generic_ptr g_ptr)`	Converts a MetaIV-IDL object reference to the corresponding 'real' MetaIV C++ value.

Name	Description
`VDM::Generic_ptr FromCPPValue(ClientID id, Generic g, VDMFactory_ptr fact);`	Converts a 'real' MetaIV C++ value to a `VDMGeneric` CORBA object that can be passed in calls to the server. Notice that you must pass to this function a handle to the VDMFactory as well.

6.4.2 Naming Conventions

To create a reference to an IDL interface *I* in C++, a variable of type `I_var` should be created.

To access an operation *O* defined in an interface *I*, indirect access of the object reference is used i.e. `I_var->M`. Such references should be used for both "in" (value) parameters and "out" (result) parameters.

6.4.3 Casting

For each interface *I*, the corresponding C++ class `I` contains a static function called `_narrow`.

The `I::_narrow` function takes an argument of type `CORBA::Object_ptr` and returns a new object reference of the class `I`. Any object which may be communicated with the ORB has type `CORBA::Object_ptr`.

If the actual (runtime) type of the argument object reference can be narrowed to `I`, `I::_narrow` will return a valid object reference. Otherwise it will return a nil object reference.

6.5 Java API

In this Section we briefly describe the translation of the IDL interfaces described in Sections 6.1 and 6.2 into Java. This is based on the translation generated by the IDL To Java Compiler (Version 1.3). Note that the documentation generated by javadoc is included with the Toolbox distribution in `api/corba/javaapi-doc`.

For each interface described in Sections 6.1 and 6.2 there is a corresponding Java class in the package `jp.co.csk.vdm.toolbox.api`. Methods defined in the interfaces have the same name in the corresponding Java class. “In” parameters (value parameters) to methods are passed as values of the corresponding class; “out” parameters (result parameters) are passed as *holder* objects – see Section 6.5.2.

In addition to these classes and those described below, there is one further class (also in the package `jp.co.csk.vdm.toolbox.api`) - `ToolboxClient`. This class provides two methods

Name	**Description**
`String readRefFile()`	Returns the contents of the `[vdm|vpp]ref.ior` file.
`public VDMApplication getVDMApplication(String[] args, ToolType toolType)`	Establishes the connection to the Toolbox (depending on the toolType, either to the `SL_TOOLBOX` or the `PP_TOOLBOX`. Takes as arguments (args) respectively any command-line arguments for the ORB.

In addition, to use the API the package `org.omg.CORBA` distributed with the Java Development Kit should be used.

Note that the property `VDM_OBJECT_LOCATION` is used by `readRefFile`, but since Java does not allow normal environment variables, this value must be passed using the `-D` flag at runtime. For instance

```
java -DVDM_OBJECT_LOCATION=/tmp <java class>
```

6.5.1 Helper Classes

For each of the interfaces described in Sections 6.1 and 6.2 named *C* there is a corresponding *Helper* class named *CHelper*. From the programmer’s point of view, the main use of these classes is the provision of a *narrow* method which narrows (casts) an arbitrary CORBA object into an object of class *C*. If such a narrowing is not possible the exception `org.omg.CORBA.BAD_PARAM` is raised.

For class *CHelper* the corresponding narrowing function would be declared using the scheme

```
public static C narrow(org.omg.CORBA.Object that)
```

where `C` would be replaced by the name of the particular class.

For the following classes a narrow method is not provided as it is not meaningful:

`ClassList`
`ErrorList`
`FileList`
`ModuleList`
`ModuleStatus`
`ToolType`
`_Error`

6.5.2 Holder Classes

For each of the interfaces described in Sections 6.1 and 6.2 named *C* there is a corresponding *Holder* class named *CHolder*. These are used to allow methods to return results by reference i.e. an object of a holder class would be passed to such a method as an argument, and the method would place its result in that object.

Each holder class has a public instance member, `value`, which is the typed value.

In addition to these holder classes there are four further holder classes:

Exception	**Type of `value` attribute**
`ClassListHolder`	`String[]`
`ErrorListHolder`	`_Error[]`
`FileListHolder`	`String[]`
`ModuleListHolder`	`String[]`

7 Recommended Reading

For further information on CORBA and the services this standard provides we refer to the omniORB3 user's guide, [LRG00], that gives an excellent introduction to this topic. More detailed information is available in selected chapters of the CORBA standard, [OMG96].

References

[LRG00] Sai-Lai Lo, David Riddoch, and Duncan Grisby. *The omniORB version 3.0 User's Guide*. AT&T Laboratories Cambridge, May 2000.

[OMG96] *The Common Object Request Broker: Architecture and Specification*. OMG, July 1996.

[SCSa] SCSK. *The VDM C++ Library*. SCSK.

[SCSb] SCSK. The vdm++ to java code generator. Technical report.

A Example Programs

A.1 The C++ Client Example

```
/***
*  * WHAT
*  *    This file is an example of how to implement a client
*  *    process that uses the CORBA API of the VDM Toolbox.
*  *
*  *    The file can be compiled with MS VC++ 6.0 on
*  *    windows NT/95 and with gcc 2.95.2 on Unix.
*  *
*  *    Use the makefile Makefile.nm if you use nmake on 98/NT and
*  *    Makefile if you compile on Linux
*  * ID
*  *    $Id: client_example.cc,v 1.27 2006/02/07 05:14:11 vdmtools Exp $
*  * AUTHOR
*  *    Ole Storm + $Author: vdmtools $
*  * COPYRIGHT
*  *    (C) 2005 CSK
***/

#include <iostream>
using namespace std;

#include <string>
// CORBA Initialisation and other stuff for omniORB4
#include "corba_client.h"

#ifdef _MSC_VER
#include <direct.h> // for getcwd
#else
#include <unistd.h> // for getcwd
#endif // _MSC_VER

char ABS_FILENAME[200];
VDM::ClientID client_id;

#define ADD_PATH(p,s) strcat(strcpy(ABS_FILENAME, p), s)
#define SORT_NUM 20
```

```
void EchoPrimes(int, ToolboxAPI::VDMInterpreter_var,
                ToolboxAPI::VDMApplication_var);
void EchoPrimes2(int, ToolboxAPI::VDMInterpreter_var,
                 ToolboxAPI::VDMApplication_var);
void ListModules(ToolboxAPI::VDMApplication_var app);

int main(int argc, char *argv[])
{
  const char * source_path = getenv("VDM_SOURCE_LOCATION");
  string sdir;
  if( source_path == NULL )
  {
    char buf[1024];
    if( getcwd( buf, sizeof( buf ) ) != NULL )
    {
#ifdef _MSC_VER
      // Convert backslash in path to forward slash as on Unix.
      for (char* s = buf; s = strchr(s, '\\'); s++) {
        *s = '/';
      }
#endif
      sdir = buf;
      for( int i = 0; i < 2; i++ )
      {
        unsigned int index = sdir.find_last_of( '/' );
        if( index == string::npos ) break;
        sdir = sdir.substr( 0, index );
      }
      sdir += "/";
    }
    source_path = sdir.c_str();

    cerr << "Environment variable VDM_SOURCE_LOCATION not set" << endl;
    cerr << "Default location: " << source_path << endl;
  }

  // The main handle to the VDM Toolbox:
  ToolboxAPI::VDMApplication_var app;

  // Initialise the ORB. Consult corba_client.{h,cc} and the
```

```
  // omniORB3 user maual for details on how this is done.
  init_corba(argc, argv);

  // Retrieve a handle to the VDMToolbox most recently started. The
  // handle is achieved through a string representation of a CORBA
  // object created by the VDM Toolbox. The string is written to a
  // file named object.string and located in the directory defined
  // by VDM_OBJECT_LOCATION
  // If this is not set, get_app automatically searches for the file
  // in the home (Unix) or profiles directory (Windows NT/95).

#ifdef VDMPP
  GetAppReturnCode rt = get_app(app, NULL, ToolboxAPI::PP_TOOLBOX);
#else
  GetAppReturnCode rt = get_app(app, NULL, ToolboxAPI::SL_TOOLBOX);
#endif //VDMPP
  switch(rt){
  case VDM_OBJECT_LOCATION_NOT_SET:
    cerr << "Environment variable VDM_OBJECT_LOCATION not set" << endl;
    exit(0);
  case OBJECT_STRING_NON_EXISTING:
      cerr << "The file " + GetIORFileName() + " could not be located. \
                Make sure the Toolbox is running" << endl;
    exit(0);
  case CORBA_ERROR:
    cerr << "Unable to setup the CORBA environment" << endl;
    exit(0);
  case CORBA_SUCCESS:
  default:
    break;
  }

  try{
    // Register the client in the Toolbox:
    client_id = app->Register();

    // First we acquire a handle to the VDMProject interface to
    // configure the current project:
    ToolboxAPI::VDMProject_var prj = app->GetProject();

    prj->New();  // New project
```

```
// Configure the project to contain the necessary files. The
// files must be located in the same directory as where the
// VDM Toolbox was started. Otherwise the absolute path to the
// files should be used
if(app->Tool() == ToolboxAPI::SL_TOOLBOX)
  {
    prj->AddFile(ADD_PATH(source_path,"examples/sort/sort.vdm"));
  }
else{
  prj->AddFile(ADD_PATH(source_path,"examples/sort/implsort.vpp"));
  prj->AddFile(ADD_PATH(source_path,"examples/sort/sorter.vpp"));
  prj->AddFile(ADD_PATH(source_path,"examples/sort/explsort.vpp"));
  prj->AddFile(ADD_PATH(source_path,"examples/sort/mergesort.vpp"));
  prj->AddFile(ADD_PATH(source_path,"examples/sort/sortmachine.vpp"));
}
// Parse the files:
ToolboxAPI::VDMParser_var parser = app->GetParser();
ToolboxAPI::FileList_var fl;
prj->GetFiles(fl);

// Parse the files in two different ways. First we traverse
// the list of files and parse each file individually. (OK, I
// know that for the SL_TOOLBOX there is only one file
// configured, but it is fine for an illustration)
cout << "Parsing files individually" << endl;
for(unsigned int i=0; i<fl->length(); i++){
  cout << (char *)fl[i] << "...Parsing...";
  if(parser->Parse(fl[i]))
    cout << "done." << endl;
  else
    cout << "error." << endl;
}

// And then we parse all files in one go:
cout << "\nParsing entire list...";
parser->ParseList(fl);
cout << "done." << endl;

// If errors were encountered during the parse they can now be
// inspected:
```

```
ToolboxAPI::VDMErrors_var errhandler = app->GetErrorHandler();
// The error handler
ToolboxAPI::ErrorList_var errs;

// retrieve the sequence of errors
int nerr = errhandler->GetErrors(errs);
if(nerr){
  // Print the error:
  cout << nerr << " errors:" << endl;
  for(int ierr=0; ierr<nerr; ierr++)
    cout << (char *) errs[ierr].fname << ", "
         << errs[ierr].line << endl
         << (char *) errs[ierr].msg << endl;

}
// Warnings can be queried similarly.

// List the names and status of all modules:
ListModules(app);

// Type check all modules:
ToolboxAPI::VDMTypeChecker_var tchk = app->GetTypeChecker();
ToolboxAPI::ModuleList_var modules;
prj->GetModules(modules);
cout << "Type checking all modules...";
if(tchk->TypeCheckList(modules))
  cout << "done." << endl;
else
  cout << "errors." << endl;

// List the new status of all modules:
ListModules(app);


// Finally we will show how to use the interpreter.

cout << endl << "Interpreter tests:" << endl << endl;
ToolboxAPI::VDMInterpreter_var interp = app->GetInterpreter();


// Call a function that computes primes:
```

```
EchoPrimes(20, interp, app);

// Secondly we show how to use Apply:
// Construct a sequence of integers to be sorted. To do
// so we need a handle to the VDMFactory to produce VDM values:
VDM::VDMFactory_var fact = app->GetVDMFactory();

app->PushTag(client_id); // Tag all objects created from now on

VDM::VDMSequence_var list = fact->MkSequence(client_id);
VDM::VDMNumeric_var elem;
for(int j=0; j<SORT_NUM; j++){
  elem = fact->MkNumeric(client_id, j);
  list->ImpPrepend(elem);
}
cout << "The sequence to be sorted: " << list->ToAscii() << endl;

// Construct the argument list for the call. That is, construct
// a VDM::Sequence containing all arguments in the right order:
VDM::VDMSequence_var arg_l = fact->MkSequence(client_id);
arg_l->ImpAppend(list);

// Set Verbose to true, to show the results of using the
// interpreter in the user interface:
interp->Verbose(true);
interp->Debug(true);

// First initialise the interpreter
interp->Initialize();

VDM::VDMGeneric_var g;
if(app->Tool() == ToolboxAPI::SL_TOOLBOX){
  g = interp->Apply(client_id, "MergeSort", arg_l);
}
else{ // PP_TOOLBOX
  // First we create the main sort object:
  interp->EvalCmd("create o := new SortMachine()");

  // Next, the GoSorting method is called on this object:
  g = interp->Apply(client_id, "o.GoSorting", arg_l);
}
```

```
    cout << "The sorted sequence: " << g->ToAscii() << endl;

    // Finally we iterate through the returned sequence to compute
    // the sum of all the elements of the sequence:

    VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
    int sum=0;
    for(int k=1; k<=s->Length(); k++){
      VDM::VDMNumeric_var n = VDM::VDMNumeric::_narrow(s->Index(k));
      sum += (Int(GetCPPValue(n))).GetValue();
    }
    cout << "The sum of all the elements: " << sum << endl;

    EchoPrimes2(50, interp, app);

    app->DestroyTag(client_id);

    // Unregister the client:
    app->Unregister(client_id);
  }
  catch(ToolboxAPI::APIError &ex){
    cerr << "Caught API error " << (char *)ex.msg << endl;
  }
  catch(CORBA::COMM_FAILURE &ex) {
    cerr << "Caught system exception COMM_FAILURE, \
             unable to contact server"
         << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB3 fatalException" << endl;
  }

  return 0;
}


void EchoPrimes(int n, ToolboxAPI::VDMInterpreter_var interp,
                ToolboxAPI::VDMApplication_var app)
  // Generates the sequence of primes below n and echoes the
  // sequence to stdout.
```

```
{
  app->PushTag(client_id);

  interp->Initialize ();

  // This VDM::Generic is used to hold the result from the
  // interpreter.
  VDM::VDMGeneric_var g;

  // Use EvalExpression to compute the primes below 20
  char expr[200];
  sprintf(expr, "[e|e in set {1,...,%d} \
                & exists1 x in set {2,...,e} & e mod x = 0 ]", n);
  g = interp->EvalExpression(client_id, expr);
  if(g->IsSequence()){
    cout << "All primes below " << n << ":" << endl
         << g->ToAscii() << endl;
  }
  VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
  int sum=0;
  for(int k=1; k<=s->Length(); k++){
    VDM::VDMNumeric_var n = VDM::VDMNumeric::_narrow(s->Index(k));
    sum += (Int(GetCPPValue(n))).GetValue();
  }
  cout << "The sum of all the primes: " << sum << endl;
  app->DestroyTag(client_id); // Clean up...
}

void EchoPrimes2(int n, ToolboxAPI::VDMInterpreter_var interp,
                 ToolboxAPI::VDMApplication_var app)
  // Generates the sequence of primes below n and echoes the
  // sequence to stdout.
  // Additionally this function shows how GetCPPValue can be used
  // to transfer an entire VDM value from the toolbox to the client
  // and convert it to a "real" C++ value as declared in metaiv.h
{
  // This VDM::VDMGeneric is used to hold the result from the
  // interpreter.
  VDM::VDMGeneric_var g;

  // Use EvalExpression to compute the primes below 20
```

```
  char expr[200];
  sprintf(expr, "[e|e in set {1,...,%d} & \
                exists1 x in set {2,...,e} & e mod x = 0 ]", n);
  g = interp->EvalExpression(client_id, expr);

  // Convert the VDM::Generic g into a "real" metaiv-Sequence
  // value:
  Sequence s(GetCPPValue(g));

  // Now we can safely destroy g since the entire value has been
  // transferred to the client:
  g->Destroy();

  cout << "All primes below " << n << ":" << endl << wstring2string(s.ascii()) <<
  int i, sum=0;
  Generic gg;
  for(i = s.First(gg); i; i = s.Next(gg)){
    sum += (int)Real(gg).GetValue();
  }
  cout << "The sum of all the primes: " << sum << endl;
}


void ListModules(ToolboxAPI::VDMApplication_var app)
  // This function lists the modules and their status.
{
 // The project handle
  ToolboxAPI::VDMProject_var prj = app->GetProject();

  // The Module Repository
  ToolboxAPI::VDMModuleRepos_var repos = app->GetModuleRepos();

  ToolboxAPI::ModuleList_var ml;
  prj->GetModules(ml);
  cout << "Modules:" << endl;
  for(unsigned int i=0; i<ml->length(); i++){
    // This struct is used to hold the status of a module:
    ToolboxAPI::ModuleStatus stat;
    // Get the status of the i'th module
    repos->Status(stat, ml[i]);
    // Print the status. 0 = none, 1 = OK
```

```
    cout << (int) stat.SyntaxChecked
         << (int) stat.TypeChecked
         << (int) stat.CodeGenerated
         << (int) stat.PrettyPrinted
         << " " << (char *)ml[i] << endl;
  }
}
```

A.2 The Java Client Example

```
import org.omg.CORBA.*;
import java.io.*;

import jp.co.csk.vdm.toolbox.api.ToolboxClient;
import jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.*;
import jp.co.csk.vdm.toolbox.api.corba.VDM.*;

public class client_example
{
  private static short client;
  private static VDMApplication app;

  private static final String VdmToolboxHomeWin=
      "C:\\Program Files\\The VDM-SL Toolbox v3.7.1\\examples";
  private static final String VppToolboxHomeWin=
      "C:\\Program Files\\The VDM++ Toolbox v6.7.17\\examples";
  private static final String VdmToolboxHome=
      "/home/vdm/toolbox/examples/sl";
  private static final String VppToolboxHome=
      "/home/vdm/toolbox/examples/pp";

  public static void main(String args[])
  {
    try {
      //
      // Create ORB
      //
```

```
    String ToolboxHome = System.getProperty("TOOLBOXHOME");
    if (System.getProperty("VDMPP") == null) {
        app = (new ToolboxClient ()).getVDMApplication(args,
                                                  ToolType.SL_TOOLBOX);

if( null == ToolboxHome ) {
          if (System.getProperty("WIN") == null)
              ToolboxHome = VdmToolboxHome;
          else
            ToolboxHome = VdmToolboxHomeWin;
        }
    }
    else {
        app = (new ToolboxClient ()).getVDMApplication(args,
                                                  ToolType.PP_TOOLBOX);

if( null == ToolboxHome ) {
          if (System.getProperty("WIN") == null)
            ToolboxHome = VppToolboxHome;
          else
            ToolboxHome = VppToolboxHomeWin;
        }
    }

    // Register the client in the Toolbox:

    client = app.Register();

    System.out.println ("registered: " + client);

    // First we acquire a handle to the VDMProject interface to
    // configure the current project:

    try{
      VDMProject prj = app.GetProject();
      prj.New();

      // Configure the project to contain the necessary files.
      // The files must be located in the same directory as where
      // the VDM Toolbox was started. Otherwise the absolute path
      // to the files should be used
```

```
if(app.Tool() == ToolType.SL_TOOLBOX){
  prj.AddFile(ToolboxHome + "/sort/sort.vdm");
}
else{
  prj.AddFile(ToolboxHome + "/sort/implsort.vpp");
  prj.AddFile(ToolboxHome + "/sort/sorter.vpp");
  prj.AddFile(ToolboxHome + "/sort/explsort.vpp");
  prj.AddFile(ToolboxHome + "/sort/mergesort.vpp");
  prj.AddFile(ToolboxHome + "/sort/sortmachine.vpp");
}

// Parse the files:
VDMParser parser = app.GetParser();
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;

// Parse the files in two different ways. First we traverse
// the list of files and parses each file individually.
// (OK, I know that for the SL_TOOLBOX there is only one
// file configured, but it is fine for an illustration)

System.out.println("Parsing files individually");
for(int i=0; i<flist.length; i++){
  System.out.println(flist[i]);
  System.out.println("...Parsing...");
  if(parser.Parse(flist[i]))
    System.out.println("done.");
  else
    System.out.println("error.");
}

// And then we parse all files in one go:

System.out.println("Parsing entire list...");
parser.ParseList(flist);
System.out.println("done.");

// If errors were encountered during the parse they can now
```

```
// be inspected:

// The error handler

VDMErrors errhandler = app.GetErrorHandler();

ErrorListHolder errs = new ErrorListHolder();
// retrieve the sequence of errors
int nerr = errhandler.GetErrors(errs);
 jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.Error errlist[] =
   errs.value;
if(nerr>0){
  // Print the errors:
  System.out.println("errors: ");
  for(int i=0; i<errlist.length; i++){
    System.out.println(errlist[i].fname);
    System.out.println(errlist[i].line);
    System.out.println(errlist[i].msg);
  }
}

// Warnings can be queried similarly.

// List the names and status of all modules:
ListModules(app);

// Type check all modules:

VDMTypeChecker tchk = app.GetTypeChecker();
ModuleListHolder moduleholder = new ModuleListHolder();
prj.GetModules(moduleholder);
String modules[] = moduleholder.value;
System.out.println("Type checking all modules...");
if(tchk.TypeCheckList(modules))
  System.out.println("done.");
else
  System.out.println("errors.");

// List the new status of all modules:
ListModules(app);
```

```
// Finally we will show how to use the interpreter.

System.out.println("Interpreter tests:");

VDMInterpreter interp = app.GetInterpreter();


// Call a function that computes primes:
EchoPrimes(20, interp, app);

// Secondly we show how to use Apply:
// Construct a sequence of integers to be sorted. In order
// to do so we need a handle to the VDMFactory to produce
// VDM values:

VDMFactory fact = app.GetVDMFactory();

app.PushTag(client); // Tag all objects created from now on

VDMSequence list = fact.MkSequence(client);

VDMNumeric elem;
for(int j=0; j<20; j++){
  elem = fact.MkNumeric(client, j);
  list.ImpPrepend(elem);
}

System.out.println("The sequence to be sorted: " +
                   list.ToAscii());

// Construct the argument list for the call. That is,
// construct a Sequence containing all arguments in the
// right order:
VDMSequence arg_l = fact.MkSequence(client);

arg_l.ImpAppend(list);

// Set Verbose to true, to show the results of using the
// interpreter in the user interface:
```

```
interp.Verbose(true);
interp.Debug(true);

// First initialise the interpreter
System.out.println("About to initialize the interpreter");
interp.Initialize();

VDMGeneric g;
if(app.Tool() == ToolType.SL_TOOLBOX){
  g = interp.Apply(client, "MergeSort", arg_l);
}
else{ // PP_TOOLBOX
  // First we create the main sort object:
  interp.EvalCmd("create o := new SortMachine()");

  // Next, the GoSorting method is called on this object:
  g = interp.Apply(client, "o.GoSorting", arg_l);
}

System.out.println("The sorted sequence: " + g.ToAscii());

// Finally we iterate through the returned sequence to
// compute the sum of all the elements of the sequence:

VDMSequence s = VDMSequenceHelper.narrow(g);

VDMGenericHolder eholder = new VDMGenericHolder();

int sum=0;
for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
  VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
  sum = sum + GetNumeric( num );
}

System.out.println("The sum of all the elements: " +  sum);

EchoPrimes2(50, interp, app);

app.DestroyTag(client);

app.Unregister(client);
```

```
      System.exit(0);
    }
    catch(APIError err) {
      System.err.println("API error"+err.getMessage ());
      System.exit(1);
    }
  }
  catch
    (jp.co.csk.vdm.toolbox.api.ToolboxClient.CouldNotResolveObjectException ex)
    {
      System.err.println(ex.getMessage());
      System.exit(1);
    }
  catch(COMM_FAILURE ex) {
    System.err.println(ex.getMessage());
    ex.printStackTrace();
    System.exit(1);
  }
};

public static void ListModules(VDMApplication app){

  try{
    // This function lists the modules and their status.
    // The project handle

    VDMProject prj = app.GetProject();

    // The Module Repository
    VDMModuleRepos repos = app.GetModuleRepos();

    ModuleListHolder ml = new ModuleListHolder();
    prj.GetModules(ml);
    String mlist[] = ml.value;
    System.out.println("Modules:");

    for(int i=0; i<mlist.length; i++){

      // This struct is used to hold the status of a module:
      ModuleStatusHolder stateholder = new ModuleStatusHolder();
      // Get the status of the i'th module
```

```
      repos.Status(stateholder, mlist[i]);
      ModuleStatus stat = stateholder.value;

      // Print the status.
      System.out.println(mlist[i]);
      System.out.println("SyntaxChecked: " + stat.SyntaxChecked);
      System.out.println("TypeChecked: " + stat.TypeChecked);
      System.out.println("Code generated: " + stat.CodeGenerated);
      System.out.println("PrettyPrinted: " + stat.PrettyPrinted);
    }
  }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}


public static void EchoPrimes(int n, VDMInterpreter interp,
                                VDMApplication app)
{
  try{
    // Generates the sequence of primes below n and echoes the
    // sequence to stdout.

    app.PushTag(client);

    // This Generic is used to hold the result from the
    // interpreter.
    VDMGeneric g;

    // Use EvalExpression to compute the primes below 20

    String expr = "[e|e in set {1,...,"+n+"} &"+
                  " exists1 x in set {2,...,e} & e mod x = 0]";
    g = interp.EvalExpression(client,expr);

    if(g.IsSequence()){
      System.out.println("All primes below " + n + ": " +
                         g.ToAscii());
    }
```

```
    VDMSequence s = VDMSequenceHelper.narrow(g);

    VDMGenericHolder eholder = new VDMGenericHolder();

    int sum=0;
    for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
      VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
      sum = sum + GetNumeric( num );
    }
    System.out.println("The sum of all the primes: " + sum);
    app.DestroyTag(client); // Clean up...
  }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}

public static void EchoPrimes2(int n, VDMInterpreter interp,
                               VDMApplication app)
{
  // Generates the sequence of primes below n and echoes the
  // sequence to stdout.
  // Additionally this function shows how GetCPPValue can be used
  // to transfer an entire VDM value from the toolbox to the
  // client and convert it to a "real" Java value as declared in
  // jp.co.csk.vdm.toolbox.VDM

  try{
    app.PushTag(client);

    // This VDMGeneric is used to hold the result from the
    // interpreter.
    VDMGeneric g;

    // Use EvalExpression to compute the primes below 20

    String expr = "[e|e in set {1,...,"+n+"} &" +
                  " exists1 x in set {2,...,e} & e mod x = 0]";
    g = interp.EvalExpression(client,expr);
```

```
    if(g.IsSequence()){
      System.out.println("All primes below " + n + ": " + g.ToAscii());
    }

    VDMSequence s = VDMSequenceHelper.narrow(g);

    // Conversion to real Java VDM value!

    java.util.LinkedList sj =
        new java.util.LinkedList ();

    VDMGenericHolder eholder = new VDMGenericHolder();

    // Convert the Generic g into a "real" Java Sequence value

    for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
      VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
      sj.add(new Integer( GetNumeric( num ) ));
    }

    int sum=0;
    for (java.util.Iterator itr = sj.iterator();
         itr.hasNext();){
        Integer i = (Integer) itr.next ();
        sum = sum + i.intValue();
    }

    System.out.println("The sum of all the primes: " + sum);
    app.DestroyTag(client); // Clean up...
  }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}

public static int GetNumeric( VDMNumeric num )
{
  byte[] b1 = num.GetCPPValue();
  try
```

```
    {
      InputStream is = new ByteArrayInputStream( b1 );
      int type = is.read();
      int c = -1;
      int last = -1;
      String str = "";
      while( true )
      {
        c = is.read();
        if ( ( c == -1 ) || ( c == ',' ) )
        {
          last = c;
          break;
        }
        str += Character.toString( (char)c );
      }
      return Integer.parseInt( str );
    }
    catch( Exception e )
    {
      return 0;
    }
  }
}
```